

## Testing de Software

### Teórico

Parte 1 – <https://youtu.be/CkRfwRR5KJM> ✓

Parte 2 – <https://youtu.be/MQPfWdtGYfw> ✓

Parte 3 – <https://youtu.be/Oggb1RqDlaE> ✓

### Prueba de Software en contexto



Cuando vimos gestión de configuración, hablamos del contexto del aseguramiento de la calidad, y dentro de este contexto teníamos calidad de proceso, calidad de producto y **prueba de software o testing**.

### Testing en el contexto: Asegurar la calidad vs controlar la calidad

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:

- La calidad no puede “inyectarse” al final
- La calidad del producto depende de tareas realizadas durante todo el proceso
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos
- La calidad no solamente abarca aspectos del producto sino también del proceso y como estos se pueden mejorar, que a su vez evita defectos recurrentes.
- El testing NO puede asegurar ni calidad en el software ni software de calidad.

Una cosa muy importante a tener en cuenta a la hora de hablar de testing, es que el testing no es el responsable, o no es la actividad que nos permite ni asegurar la calidad, ni desarrollar un software de calidad.

El testing es una disciplina en la que nosotros estamos contemplando aspectos que tienen que ver con el producto y con los requerimientos de calidad, pero NO es la actividad que nos permite asegurarnos que nuestro proceso para construir el software sea de calidad.

¿Por qué? Por un lado, porque la calidad del software no abarca solamente aspectos del producto, si nos concentramos solamente en aspectos del producto, estamos viendo una parte de lo que tiene que ver con la calidad del software. Por el otro lado, porque hay otras disciplinas o actividades que incluso se ejecutan en términos de la línea, proceso o ciclo de vida, antes del testing y que nos permiten asegurarnos la calidad en el producto que vamos a construir. Estas actividades tienen que ver con la inspección o con revisiones técnicas (escapan de las pruebas de software).

Entonces, no hay que imaginarnos al testing como una manera de asegurarnos de construir software de calidad.

### ¿Por qué el testing es necesario? - Quick Quiz

- **Porque la existencia de defectos en el software es inevitable** ✓  
El testing es un proceso que iniciamos asumiendo que va a haber defectos en el software. Siempre partimos de esta premisa, los defectos en el software existen y son inevitables, porque el software está desarrollado por personas. Nuestro objetivo cuando nos planteamos hablar de testing o ejecutar las actividades de testing tiene que ver con que sabemos que vamos a encontrar defectos en el software.
- **Para llenar el tiempo entre fin de desarrollo y el día del release** ✗  
Claramente NO
- **Para probar que no hay defectos** ✗  
Está en contraposición con la primera afirmación. Si decimos que la existencia de defectos del software es inevitable, claramente el fin del testing NO es probar que no hay defectos, porque los hay.
- **Porque el testing está incluido en el plan del proyecto** ✗  
No es así, el testing no se incluye como un requerimiento, sino que al revés, yo decido hacer testing y lo incluyó en un plan de proyecto porque es una actividad y un proceso que son necesarios.
- **Porque debuggear mi código es rápido y simple** ✗  
No tiene nada que ver con el testing. Por ahí hay una confusión entre un nivel de testing, pero no se relaciona con el testing.
- **Para evitar ser demandado por mi cliente** ✓
- **Para reducir riesgos** ✓
- **Para construir la confianza en mi producto** ✓
- **Porque las fallas son muy costosas** ✓
- **Para verificar que el software se ajusta a los requerimientos y validar que las funciones se implementan correctamente** ✓

Estas son las razones por las cuales ejecutar las actividades relacionadas con el proceso del testing son necesarias.

### Romper mitos

- **El testing es una etapa que comienza al terminar de codificar** ✗  
El testing puede empezar mucho antes, incluso cuando empiezo a definir los requerimientos o desde el momento en que empiezo a definir el plan de mi proyecto.
- **El testing es probar que el software funciona** ✗  
No, porque el testing es un proceso destructivo cuyo objetivo es encontrar defectos, y NO probar que el software funcione.
- **TESTING = CALIDAD DE PRODUCTO** ✗  
Testing NO es igual a calidad de producto porque va mucho más allá de lo que es simplemente el testing.
- **TESTING = CALIDAD DE PROCESO** ✗  
Ya dijimos que NO es lo mismo.
- **El tester es el enemigo del programador** ✗

No, pero en esta cuestión de cuáles son los objetivos de la actividad de implementación vs. actividad del testing, si esto no está planteado desde un lugar de sinergia y trabajo colaborativo, por ahí se pueden entender que las actividades de ambos roles están enfrentadas.

### Definición de Prueba de Software

Visión más apropiada del Testing:

- Proceso **destructivo** de tratar de encontrar defectos (cuya presencia se asume!) en el código.  
Es destructivo porque su objetivo es encontrar defectos, es “romper” el software. Estos defectos que intentamos encontrar tienen que ver con que se asume que existen estos defectos. Justamente porque la presencia de defectos en el software es inevitable.
- Se debe ir con una **actitud negativa** para demostrar que algo es incorrecto.  
Es también un proceso destructivo, porque la actitud del testing es negativa. No me pongo en la actitud positiva de decir: “Voy a probar que el software funcione”, sino que digo: “Tengo que encontrar los defectos que ya se que estan en el código”
- Testing exitoso → es el que encuentra defectos!  
Un testing exitoso es aquel que encuentra defectos, si no encuentra defectos y partimos de la base de la definición que asume que los defectos existen en el código que estamos probando, entonces el testing NO es exitoso.
- Mundialmente: 30 a 50% del costo de un software confiable.  
Entre el 30 y el 50% del costo de un software confiable se nos va en testing. Cuando pensamos en una estimación de esfuerzo relacionada con el testing, y asociado a ese esfuerzo el costo, el testing se lleva esa parte del esfuerzo de construir un software confiable.

### Principios del Testing

- El testing muestra la presencia de defectos.  
El testing busca, muestra los defectos.
- El testing exhaustivo es imposible.  
Esta es la base de muchas cosas que vamos a trabajar. NO es viable en términos de costo, esfuerzo y tiempo, probar el software de manera completa. El objetivo es abarcar la mayor cantidad de casos posibles dentro del testing con el menor esfuerzo posible. La premisa es que nunca vamos a poder abarcar el 100%, nunca vamos a poder probar el software de manera completa.
- Testing temprano  
No es solamente lo que nos imaginamos cómo probar el software con el software funcionando, sino que el testing empieza desde el momento en que comienzo a escribir los casos de prueba, y desde el momento que lo planifico y defino cuál va a ser el nivel de cobertura del testing.
- Agrupar los defectos  
Agrupar los defectos en función del universo que voy probando, y de los casos de prueba que voy definiendo.
- Paradoja de Pesticida

A medida que vayamos ejecutando el testing del software, y a medida que vayamos ejecutando los distintos ciclos de vida, nos vamos a encontrar con la Paradoja del Pesticida. Pasa algo parecido a lo que pasa cuando uno usa muchas veces un pesticida, donde probablemente no tengamos problemas con el insecto o la plaga que quiero matar pero sí con otras que no las estoy viendo, porque estoy concentrada en la plaga que el pesticida puede eliminar.

Con el testing quizás logre armar casos de pruebas que se ejecuten de manera correcta, correrlos y eliminar todos los defectos, y dejar cosas afuera que no estoy viendo porque me concentro en los defectos que trato en ese contexto. Esto ocurre mucho cuando hablamos de testing automatizado, y es por eso que a veces el testing automatizado va de la mano con un testing manual.

**Pregunta:** ¿Es posible que la actividad de testing introduzca errores? Si. La corrección está contemplada dentro del testing. Esto de testear, encontrar defectos, corregirlos y volver a testear es todo parte del testing. Hasta cuando? Hasta que decidamos dejar de testear. ¿Cómo sabemos cuándo dejar de testear? Lo vamos a ver después, sabiendo como principio que no vamos a poder testear todo, el 100% de cobertura es imposible.

El testing no termina cuando encuentro el defecto, sino que termina cuando lo corrijo o cuando termino los ciclos de prueba, alcance mi nivel de cobertura y decido que los defectos que tengo no tienen la suficiente severidad como para ejecutar un ciclo de prueba más, entonces asumo que los dejo.

- El testing es dependiente del contexto
- Falacia de la ausencia de errores
- Un programador debería evitar probar su propio código  
Porque es difícil tratar de encontrar defectos en lo que yo hice. Hay una excepción, a un nivel de testing (el primer nivel = testing unitario), donde el programador sí prueba su propio código.
- Una unidad de programación no debería probar sus propios desarrollos.  
Ningún programador debería probar su propio código, y ninguna unidad de programación debería probar sus propios desarrollos, por razones fundamentales de intentar encontrar un error dentro de un trabajo que yo hice.
- Examinar el software para probar que no hace lo que se supone que debería hacer es la mitad de la batalla, la otra mitad es ver que hace lo que no se supone que debería hacer.

Cuando pensamos en el testing hay algo que es importante. Ya sabemos que el testing es un proceso destructivo cuyo objetivo es encontrar los defectos. Cuando pensamos hacia donde apuntamos para encontrar esos defectos, tenemos que tener en cuenta que no solamente tenemos que buscar los defectos pensando en que el software haga lo que se supone que debe hacer, sino que también hay que mirar que el software no haga lo que no se supone que debe hacer.

Hay que ver ambas cosas, la prueba por el lado positivo y también por el lado negativo.

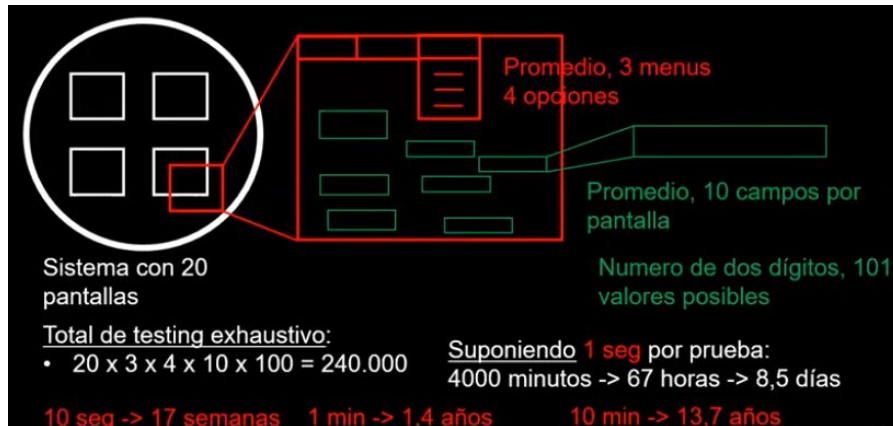
- No planificar el esfuerzo de testing sobre la suposición de que no se van a encontrar defectos.

Nunca debemos planificar el esfuerzo del testing pensando o suponiendo que no vamos a encontrar defectos, porque asumimos que los defectos están.

### ¿Cuanto testing es suficiente?

El testing exhaustivo NO es posible. Como se cuando dejo de probar.

#### Ejemplo - Sistema con 20 pantallas (pequeño)



- Tenemos 3 menús de 4 opciones por cada pantalla
- Tenemos 10 campos por cada pantalla
- Dentro de los campos tenemos números de 10 dígitos y de 101 valores posibles por cada campo

En términos de testing exhaustivo, si suponemos que cada caso de prueba nos va a llevar un minuto, tendríamos exponencialmente un año y medio de una persona probando el software de manera exhaustiva. Claramente nos podemos dar cuenta que esto no es viable, y aca no incluimos lo que es el **test de regresión**, que es que si cuando yo pruebo el software, la corrección no introduce nuevos defectos.

En este contexto, y sabiendo que el testing exhaustivo no es posible ¿Cómo hacemos? Evidentemente tenemos que asegurarnos encontrar la mayor cantidad posible de defectos y poder corregirlos, para evitar ser demandado por mi cliente, para tener confianza en el producto que entrego, para minimizar los riesgos, etc.

### ¿Cuanto testing es suficiente? Quick Quiz

- **Cuando se tiene la confianza de que el sistema funciona correctamente** ✓  
En principio, lo que nosotros tenemos que tener en cuenta es que para que sepamos o consideremos que hemos llegado a realizar la cantidad de pruebas o la cantidad de ciclos y casos de pruebas suficiente, los hemos ejecutado, es cuando tenemos la confianza de que mi software funciona correctamente. ¿Cómo medimos eso? Tengo la confianza suficiente aunque no siga testeando, sabiendo que puede haber más defectos, pero que la confianza que logre a través del testing que hice es suficiente.
- **Depende del riesgo de tu sistema** ✓  
No es lo mismo una aplicación para hacer pedidos, que un software a partir del cual se inyectan drogas a pacientes internados. Claramente el riesgo que corro permitiendo que haya defectos en uno y otro modelo son diferentes. Esto es importante a la hora de determinar hasta cuando voy a testear.
- **Cuando se termino todo lo planificado** ✗

- **Nunca es suficiente** ❌
- **Cuando se ha probado que el sistema funciona correctamente** ❌

## Conclusiones

- El testing exhaustivo es imposible
- Decidir cuánto testing es suficiente depende de:
  - Evaluación del nivel del riesgo  
Cuando determinamos hasta cuando testeamos, o cuando dejamos de testear hay que tener en cuenta la variable de nivel de exposición del riesgo. Cuanto mayor sea el riesgo que manejo, más cantidad de testing voy a tener que realizar, y por consiguiente más esfuerzo y dinero le voy a dedicar al testing.
  - Costos asociados al proyecto  
Si decimos que el testing nos consume entre el 30 y el 50% del esfuerzo de un software confiable, entre el 30 y el 50 hay una brecha. ¿Cómo definimos cuánto? Los costos del proyecto son una variable que van junto con el riesgo. Si tengo un presupuesto escaso para mi proyecto, y tengo un software cuyo uso implica un riesgo considerable, en el caso que haya defectos, el proyecto no será viable.
- Usamos los riesgos para determinar:
  - Qué testear primero  
Tengo determinado nivel de riesgo y en función de ese nivel elijo las funcionalidades o los casos de prueba que voy a testear primero.
  - A qué dedicarle más esfuerzo
  - Cuales son las cosas que por ahora podemos no testear.

Todo esto no es solamente una enunciación cualitativa. Empieza a aparecer la palabra que se usa mucho cuando armamos el plan de pruebas que es el “Criterio de aceptación”.

El **criterio de aceptación** es lo que comúnmente se usa para resolver el problema de determinar cuándo una determinada fase de testing ha sido completada.

Es algo que acuerdo con mi cliente y esto es importante, mi cliente tiene que acordar conmigo hasta donde voy a probar. Porque esto es lo que me queda de respaldo en el caso de que después se encuentren defectos. ¿Cómo defino el criterio de aceptación? Tiene que ser una variable que pueda medirse.

Puede ser definido en términos de:

- Costos  
Voy a probar o voy a usar tantas horas en el testing
- % de tests corriendo sin fallas  
Voy a ejecutar hasta que haya un determinado porcentaje de tests sin fallas. Es decir, si tengo 100 casos de prueba, voy a probar hasta que el 80% de ese número no tenga fallas. Cuando el 80% de los casos no tenga fallas, asumo que dejo de probar y que el software puede entrar en producción.
- Fallas predichas aún permanecen en el software  
Hay que recordar que cuando nosotros entreguemos el software, el software va a tener defectos. Tenemos que saber esto y también tenemos que hacérselo saber al cliente. El software va a tener defectos porque el testing no es exhaustivo. Los

defectos van a aparecer, permanecen en el software porque no tengo forma de probar el software de manera completa. Ni hablar de funcionalidades que quedan relegadas en su uso, y que en algún momento se dispara que se empiecen a utilizar.

- No hay defectos de una determinada severidad en el software

Voy a probar hasta en tanto sigan apareciendo defectos de severidad crítica o alta.

Si los defectos tienen severidad leve o menor, dejo de probar.

Son distintas formas de enunciar el criterio de aceptación, que es importante que tengamos en claro para poder acordar con el cliente y para saber hasta donde vamos a probar. Esta es la medida clave.

**Pregunta:** Entre estos 4 términos, nosotros como empresa definimos cuáles usamos? Nos ponemos de acuerdo con el cliente. Estos términos son solo ejemplos, podría ser también definir un nivel de cobertura. Son ejemplos de cómo podemos cuantificar este criterio de aceptación. Lo importante es que no quede en términos cualitativos, sino que tengamos una manera concreta de medirlo, con un número o con una condición que nosotros podamos definir si se alcanzó o no se alcanzó, sin tener ambigüedades en el medio, porque es justamente lo que acordamos con el cliente. Es importante siempre que el cliente conozca los criterios de aceptación.

**Pregunta:** Excepto por el costo, los demás son demasiado técnicos. ¿Cómo hago para explicarle al cliente este criterio? Por ejemplo, con el tema de severidad de defectos. El cliente puede entender perfectamente lo que le decimos, si somos claros en explicar cómo clasificamos los defectos de acuerdo a la severidad. Quizás el % de tests corridos sin fallas, si sea más técnico, y si nuestro cliente no tiene forma de entender un criterio de aceptación técnico, no uso ese término para definir el criterio de aceptación. Otra manera de definirlo es a través de un nivel de cobertura, voy a probar el 50% del sistema y voy a probar estas funcionalidades que considero como críticas en un tanto por ciento de nivel de cobertura, quizás esta manera sea más entendible para nuestro cliente. Esto es lo que justamente hay que trabajar y definir en función de lo que nuestro cliente pueda llegar a entender.

A esta altura es una locura pensar que cuando el software se entrega no va a haber posibilidad de que aparezcan defectos. Queda en la habilidad del líder de proyecto hablar con el cliente y explicarle como funciona el software. Cuando hablamos de framework ágiles donde el PO es parte del equipo y está más involucrado, es más fácil. Cuando hablamos de proyectos gestionados de manera tradicional, donde el cliente de alguna manera o el sponsor están afuera del proyecto, es más difícil. Siempre es mejor ser claros y explicar de qué estamos hablando, por que aparecen los defectos en el software, y no prometer algo que no vamos a poder cumplir.

Hay una variable que es fundamental para el cliente, que tiene que ver con el costo. “Para llegar hasta este nivel de cobertura el costo es este...”, aca el cliente entiende que es preferible a veces asumir algunos riesgos, dejando de probar algunas cosas que no son tan críticas. Sino, saldría más caro el testing que construir el software.

## **La Psicología del testing**

- La búsqueda de fallas puede ser visto como una crítica al producto y/o su autor

- La construcción del software requiere otra mentalidad a la hora de testear el software

La mentalidad de quien testea no es que estoy haciendo una crítica al programador, o estoy diciendo que el programador hizo mal su trabajo porque encontré defectos. La presencia de defectos existe siempre. Si el equipo comprende este contexto, nos evitamos la rivalidad entre el tester y el programador, y la cosa de pensar que es el tester el que me dice si hice bien o mal mi trabajo. La búsqueda de defectos NO es una crítica hacia el programador, es una actividad que se hace orientada a la calidad del producto que estamos construyendo. No como una crítica hacia el trabajo del desarrollador.

Probablemente cuando se trabaja en términos de framework ágiles donde todos hacen todo, y hay un espíritu más de equipo y colaborativo, se busca romper un poco con la rivalidad de los distintos roles. Cada uno tiene su rol y la presencia de defectos en el software se asume, existe, NO hay manera de construir un software, no hacerle testing, y que esos defectos no se encuentren. En este contexto, cada uno tiene un rol en la construcción del software, y cada uno de esos roles es igual de importante a la hora de construir el producto final de software que tenemos que construir.

### Conceptos: Error vs. Defectos

Cuando hablamos de testing usamos el término de “defecto” y NO de error. Cuando hablamos que en el testing encontramos defectos, esos defectos que encontramos en esa etapa de testing fueron generados en una etapa anterior, en este caso la etapa de implementación. Cuando se escribió el código se generó el error. Cuando ese error pasó a la etapa siguiente y lo detecte ahí, ese error se convierte en defecto, es decir, ya NO es más un error, sino que es un defecto.

El **error** es algo que se detecta en la misma etapa en que se genera. En cambio, el **defecto** se detecta en una etapa posterior. El testing detecta el defecto en una etapa posterior, y se genera en la etapa de implementación.

### Defectos, severidad y prioridad

#### Severidad

- 1) Bloqueante
- 2) Crítico
- 3) Mayor
- 4) Menor
- 5) Cosmético

#### Prioridad

- 1) Urgencia
- 2) Alta
- 3) Media
- 4) Baja

¿Cómo se clasifican los defectos?

Normalmente se clasifican bajo dos variables: severidad y prioridad.

Podemos tener en cuenta la **severidad** a la hora de definir el criterio de aceptación. Por ejemplo, si todos los defectos que encontré en el último ciclo de pruebas son de severidad menor o cosmética dejo de probar. Ese sería el criterio de aceptación de fin del testing. Mientras haya encontrado defectos con severidad bloqueante, crítica o mayor tengo que



seguir haciendo testing hasta que los defectos con esa severidad se eliminen. La severidad tiene que ver con qué pasa si el defecto permanece en el software.

- **Bloqueante:** si el defecto existe no puedo seguir ejecutando el caso de prueba.
- **Crítico:** a nivel de la funcionalidad que estoy ejecutando, es necesario corregir ese defecto si no la funcionalidad no va a ser adecuada
- **Mayor:** lo mismo que crítico, pero crítico es peor. Sigue siendo grave.
- **Menor:** por ejemplo, cuando estoy ejecutando una funcionalidad por un curso alternativo y no me tiene que dejar seguir avanzando y el mensaje que me muestra me dice otra cosa que no corresponde. El problema es que el mensaje hacia el usuario no es claro.
- **Cosmético:** tiene que ver con las etiquetas, con los errores de ortografía, etc.

La **prioridad** está relacionada con la prioridad de la funcionalidad y del caso de prueba en el contexto de esa funcionalidad dentro del negocio. Entonces, de acuerdo al uso del negocio y a la funcionalidad es que voy a clasificar la prioridad de tratamiento a la hora de resolver el defecto.

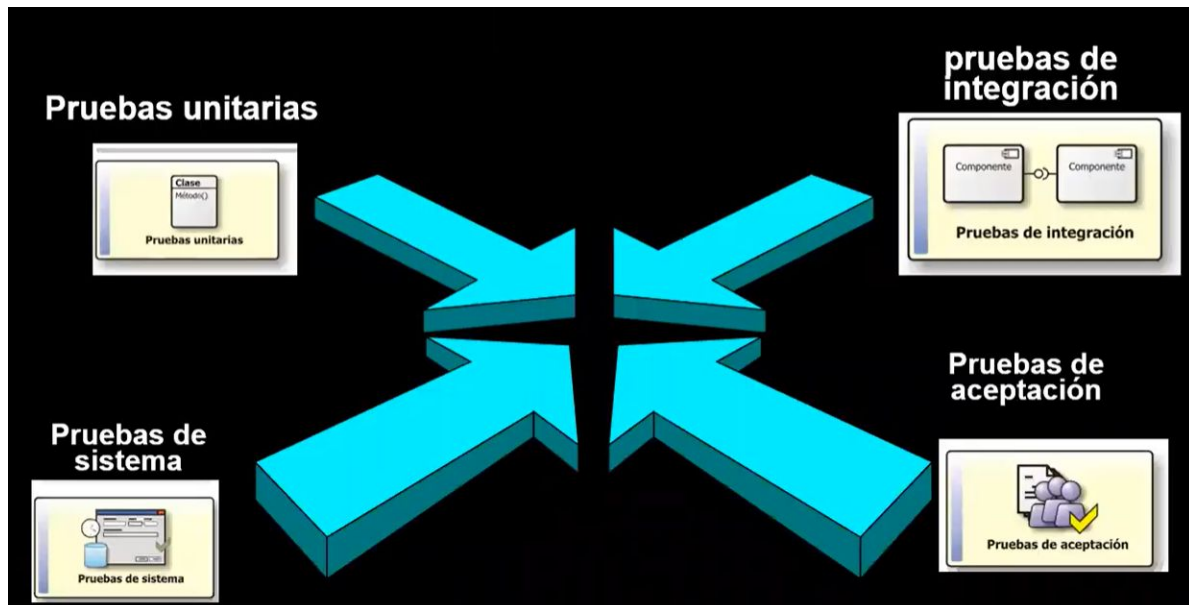
No necesariamente todo lo bloqueante es urgente. No hay una relación directa entre severidad y prioridad. Puede haber algunos defectos que tengan una alta severidad pero que en términos de las funcionalidades del negocio que son más críticas tengan una urgencia media.

El bloqueante, crítico o mayor es en el contexto del caso de prueba que estoy probando. Cuando yo defino la severidad, estoy parada en el caso de prueba y en el defecto que encontré, no estoy pensando en lo que eso significa para el negocio, sino en lo que el defecto que yo estoy encontrando significa para el caso de prueba que estoy ejecutando. Cuando hablamos de prioridad ya si estamos hablando en términos de lo que significa esa funcionalidad para el negocio.

**Pregunta:** Siempre hablamos de testing a nivel de funcionalidad, ¿no estamos contemplando cuestiones de seguridad? Si, también, ya vamos a ver que hay dos testing, uno que es funcional y otro que es no funcional. El testing tiene que incluir las dos cosas, el funcional y el no funcional.

No funcional: seguridad, performance, algunas otras características del software que tenemos que testearlas antes de que el software vaya a producción.

## **Niveles de Prueba**



Vamos a ir concretamente a cómo se hacen las pruebas.

Los **niveles de prueba** tienen que ver con cómo vamos escalando en términos de lo que vamos probando haciendo un enfoque desde la menor granularidad hasta la mayor. El primer nivel de prueba que nos aparece en este contexto es el Testing Unitario.

### Testing Unitario

- Se prueba cada componente tras su realización/construcción
- Solo se prueban componentes individuales  
Es el nivel donde hago foco sobre un componente.
- Cada componente es probado de forma independiente  
Termino de construir el componente de manera individual y lo testeo. De manera independiente, no hago ningún testeo vinculándolo con otros componentes.
- Se produce con acceso al código bajo pruebas y con el apoyo del entorno de desarrollo, tales como un framework de pruebas unitarias o herramientas de depuración.
- Los errores se suelen reparar tan pronto como se encuentran, sin constancia oficial de los incidentes.

Este es el testing que suele ejecutar el mismo programador que construyó el código. Porque muchas veces se hace con herramientas de depuración.

Es el primer testing candidato a ser automatizado. De hecho, hay algunas maneras de construir código escribiendo el testing automatizado y a partir de la ejecución del testing automatizado empezar a construir el software o escribir el código, se llama TDD.

¿Cómo funciona? Se ejecuta el testing unitario, se encuentran los errores, se corrigen, y sigue adelante.

### Testing de Integración

- Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
- Cualquier estrategia de prueba de versión o de integración debe ser incremental, para lo que existen dos esquemas principales:
  - Integración de arriba hacia abajo (top-down)
  - Integración de abajo hacia arriba (bottom-down)
- Lo ideal es una combinación de ambos esquemas
- Tener en cuenta que los módulos críticos deben ser probados lo más tempranamente posible
- Los puntos clave del test de integración son simples:
  - Conectar de a poco las partes más complejas
  - Minimizar la necesidad de programas auxiliares

Como su nombre lo indica, ya empezamos a juntar esos componentes que probamos de manera aislada. Normalmente, cuando empezamos a juntarlos, hay un esquema de integrar esos componentes de manera incremental. No es que vamos a integrar todos los componentes juntos, si no que lo vamos a ir integrando de a poco y a medida que lo vamos integrando vamos a ir probando, y a medida que esas pruebas vayan siendo exitosas, integramos nuevos componentes.

Siempre probamos e integramos primero los módulos críticos para el negocio, lo que es más importante para el negocio y a partir de ahí lo empezamos a hacer extensivo hacia las otras funcionalidades que son más secundarias o de menor prioridad.

### **Testing de Sistema**

- Es la prueba realizada cuando una aplicación está funcionando como un todo (prueba de la construcción final)
- Trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc.)
- El entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron en las pruebas
- Deben investigar tanto requerimientos funcionales y no funcionales del sistema

Aca ya tenemos el sistema de manera completa. Empezamos a ver requerimientos funcionales y no funcionales. Tratamos de probar en un ambiente que sea lo más parecido a nuestro entorno de producción. Es importante plantearnos justamente el foco no solamente en las pruebas funcionales, sino también en los requerimientos no funcionales.

### **Testing de Aceptación**

- Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades  
Ya no es un testing técnico como el que veníamos haciendo hasta ahora.
- La meta en las pruebas de aceptación es el de establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema

- Encontrar defectos no es el foco principal en las pruebas de aceptación
- Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta).

El foco cambia bastante con respecto a las pruebas que hacen los testers, en donde nuestro propósito era encontrar defectos. El usuario no va a hacer eso, no va a tener el foco en encontrar los defectos, lo que quiere ver es si el sistema se comporta o hace lo que él espera.

El ambiente de trabajo realizado por el usuario sea también lo más parecido al entorno final de producción.

- Pruebas alfa: ambiente de laboratorio, es decir, no es nuestro entorno de desarrollo, nuestro entorno de testing pero si de laboratorio
- Pruebas beta: ambiente de trabajo real

Objetivo → Que el usuario nos dé el OK

### Relación con el Framework de Scrum

¿En qué ceremonia se haría el testing de aceptación? En la **review**, donde el cliente acepta nuestro producto. Donde el PO mira las features que estamos entregando y nos dice si es lo que esperaba o no.

### Proceso de Pruebas

Cuando hablamos en términos de metodologías tradicionales o procesos definidos la prueba es también un proceso. Es un proceso que tiene distintas etapas. En Scrum esto es diferente, no ocurre este proceso, justamente porque trabaja con procesos empíricos.



- **Planificación:** cuando armamos el plan de pruebas, definimos el criterio de aceptación, es un plan subsidiario al plan de proyecto, es una partecita, así como teníamos el plan de gestión de configuración por ejemplo, también tenemos el plan de prueba.
- **Identificación y especificación de los casos de prueba:** de lo que vamos a probar, definimos qué es lo que vamos a probar.

- **Ejecución:** en donde ejecutamos esos casos de prueba que tenemos especificados anteriormente y vemos cuáles son los resultados, si hay defectos o no llegamos al criterio de aceptación que teníamos definido volvemos a ejecutar, volvemos a analizar las fallas. Hasta cuando? Hasta que logremos cumplir con nuestro criterio de aceptación, dejamos de probar y las pruebas terminan.

## Planificación y control

- La **planificación** de las pruebas es la actividad de verificar que se entienden las metas y los objetivos del cliente, las partes interesadas (stakeholders), el proyecto, y los riesgos de las pruebas que se pretenden abordar.
- Construcción del Test Plan:
  - Riesgos y Objetivos del Testing
  - Estrategias de Testing
  - Recursos
  - Criterio de Aceptación
- Controlar
  - Revisar los resultados del testing
  - Test coverage y criterio de aceptación
  - Tomar decisiones

Construimos el plan de pruebas y acordamos metas y objetivos con los clientes. Definimos la estrategia de testing, como vamos a hacer las pruebas, qué pruebas van a estar automatizadas, qué pruebas van a ser manuales, cuál va a ser el nivel de cobertura, todo esto lo definimos dentro de la planificación.

Como toda parte subsidiaria del plan de proyecto, la planificación es algo **vivo**. No se define una vez y se guarda. Se va retroalimentando de manera permanente y también puede implicar que en algún momento le haga alguna modificación.

## Identificación y Especificación

- Revisión de la base de pruebas
- Verificación de las especificaciones para el software bajo pruebas
- Evaluar la testeabilidad de los requerimientos y el sistema
- Identificar los datos necesarios
- Diseño y priorización de los casos de las pruebas
- Diseño del entorno de prueba

Las dos van de la mano. Tienen que ver con determinar qué es lo que vamos a probar, definir los casos de prueba, identificar los datos para esos casos de prueba, priorizarlos, y definir y diseñar el entorno de pruebas sobre el cual vamos a ejecutar esos casos.

Podemos ejecutar esta actividad aunque no tengamos escrita una sola línea de código. ¿Qué necesitamos? Necesitamos los requerimientos funcionales y no funcionales para poder definir cuales son nuestros casos de prueba. Lo ideal es que al menos esta actividad se desarrolle de una manera paralela para que cuando el código se termine de escribir podamos pasar a la actividad de ejecución.

## Ejecución

- Desarrollar y dar prioridad a nuestros casos de prueba
- Crear los datos de prueba
- Automatizar lo que sea necesario
- Creación de conjuntos de pruebas de los casos de prueba para la ejecución de la prueba eficientemente
- Implementar y verificar el ambiente
- Ejecutar los casos de prueba
- registrar el resultado de la ejecución de pruebas y registrar la identidad y las versiones del software en las herramientas de pruebas
- Comparar los resultados reales con los resultados esperados

Tiene que ver con ejecutar esos casos de prueba que nosotros tenemos definidos. ¿Qué implica esto? Implica no solamente ejecutar los casos de prueba, sino automatizar los casos de prueba que se puedan (vamos a empezar por los casos de prueba unitarios). Durante esta ejecución también los vamos a priorizar a los casos de pruebas. Vamos a definir cuales son los casos de prueba que constituyen un ciclo de prueba.

Cuando nosotros hicimos en la etapa anterior la identificación y especificación definimos para cada caso de pruebas, cuál era el resultado esperado. Cuando nosotros ejecutamos el testing, lo que hacemos es comprobar que los resultados que obtenemos sean o no los resultados que habíamos definido como esperados. Si los resultados que obtenemos son los esperados, el testing pasa, si los resultados que obtenemos no coinciden ese caso de prueba falla y hay que corregir y volver a ejecutar. A su vez, definiremos el defecto que se haya encontrado en función del resultado que obtuvimos, cual es su severidad y cual es su prioridad. Todo eso lo hacemos en la ejecución.

## Evaluación y Reporte

- Evaluar los criterios de aceptación
- Reporte de los resultados de las pruebas para los stakeholders
- Recolección de la información de las actividades de prueba completadas para consolidar
- Verificación de los entregables y que los defectos hayan sido corregidos
- Evaluación de cómo resultaron las actividades de testing y se analizan las lecciones aprendidas

Tiene que ver con que cuando terminamos el ciclo de prueba: ¿Cuáles son los defectos que encontramos? Estos defectos que encontramos, ¿Qué severidad tienen? Llegamos a cumplir el criterio de aceptación o todavía no? ¿Seguimos probando o no?

## Quick Quiz

**¿Cuántas líneas de código necesito para empezar a hacer testing?** Cero. Porque para todas las actividades de planificación, identificación, ejecución, preparar el ambiente de prueba no necesito escribir ninguna línea de código para empezar a hacerlas.

**Y en Scrum? ¿Qué hacemos con este proceso?** En Scrum no tenemos este proceso en estos términos. La actividad de testing queda embebida dentro de las actividades que se ejecutan en el contexto de la Sprint. El testing se hace y tiene, en términos de casos de prueba y de ejecución, las mismas características que tiene el testing cuando trabajamos con metodologías tradicionales, solamente que no vamos a tener un proceso definido con estas características si no que queda embebido dentro de la Sprint y tratando de encontrar los defectos lo antes posible y corregirlos de la manera más rápida posible.

La review era el testing de aceptación. El unitario, integración y sistema ya lo hicimos, porque si llegamos a la review con el nivel de aceptación es porque los demás ya los hicimos.

### **Caso de Prueba**

- Set de condiciones o variables bajo las cuales un tester determinará si el software está funcionando correctamente o no
- Buena definición de casos de prueba nos ayuda a REPRODUCIR defectos

Tiene que ver con escribir una secuencia de pasos que tienen condiciones y variables con las que yo voy a ejecutar el SW y me van a permitir saber si el SW está funcionando correctamente o no.

Por ejemplo, supongamos que estamos probando un SW para un GPS. Supongamos que el caso de prueba que vamos a probar es “Buscar una calle”. Cómo sería el Caso de Prueba? Primero voy a definir una serie de condiciones o variables, voy a buscar una calle que ya esté cargada en el GPS, esa es una condición que yo decido a la hora de pensar en mi caso de prueba. Entonces, supongamos que vamos a buscar la calle “San Lorenzo”, me voy a asegurar que cuando voy a ejecutar ese caso de prueba esa calle esté cargada en nuestro GPS, con una altura determinada. Entonces, ¿Cómo voy a definir ese caso de prueba? Voy a decir: “Ingreso en el GPS la calle San Lorenzo, ingreso la altura 720, pongo buscar”, el resultado esperado es que el sistema me ubique donde esta esa calle y esa altura. Entonces, ¿Qué es lo importante de la definición del caso de prueba? Primero, que tengo esas condiciones y variables determinadas claramente especificadas y tengo bien definido cual es la secuencia de pasos que tengo que ejecutar con esas variables para lograr el resultado esperado. Después puedo ejecutar y me puedo encontrar que mi SW está obteniendo el resultado esperado o no, ese será el resultado de la ejecución del caso de prueba. Pero tienen que quedar en claro cuales son todas las secuencias de pasos que el tester debe ejecutar para ver si ese resultado esperado se logra o no. Esto es fundamental porque si el resultado esperado no se logra quien tiene que reproducir un defecto para corregirlo no tiene más que seguir los pasos que están enunciados en el caso de prueba. Si yo no tengo un caso de prueba y me pongo a hacer un testing exploratorio y a probar lo que se me viene a la mente, cuando el defecto aparezca, ¿cómo voy a hacer después para reproducirlo? Puedo tener suerte y que sea fácil de reproducir o no sea fácil porque no me guíe con un caso de prueba que estaba definido. El caso de prueba es fundamental para que el tester sepa qué es lo que está testeando y sepa que es lo que ejecuta sobre todo para validar si el resultado que obtengo es el esperado y si no es, como hice para llegar a ese resultado que es diferente.

## Condiciones de prueba

- Esta es la reacción esperada de un sistema frente a un estímulo particular, este estímulo está constituido por las distintas entradas
- Una condición de prueba debe ser probada por al menos un caso de prueba

Otro aspecto es definir cuales son las condiciones de la prueba. En el ejemplo anterior, la condición de prueba era que la calle San Lorenzo ya esté cargada en el GPS. Estas condiciones de prueba son fundamentales, porque no es lo mismo que ponga como condición que la calle va a estar cargada a que ponga como condición que la calle no va a estar cargada. El resultado esperado va a ser otro, "No se encuentra el destino".

La condición de prueba es muy importante a la hora de definir los casos de prueba.

## Características de los casos de pruebas

*Un caso de prueba es la unidad de la actividad de la prueba*

Consta de tres partes:

1. **Objetivo:** la característica del sistema o comprobador  
Tiene que ver con que voy a probar concretamente
2. **Datos de entrada y de ambiente:** datos a introducir al sistema que se encuentra en condiciones preestablecidas  
Datos, condiciones previas o precondiciones, son los datos que voy a tener que contemplar, por ejemplo, cuáles son los datos y la altura que van a estar cargadas
3. **Comportamiento esperado:** la salida o la acción esperada en el sistema de acuerdo a los requerimientos del mismo  
Luego de ejecutar cada uno de los pasos que voy a ejecutar, después de eso ver cual es el ejemplo esperado.  
En el ejemplo de recién, puedo tener dos casos de prueba, uno en el que ponga de precondición que la calle San Lorenzo al 700 va a estar cargada, enunció todos los pasos:

- entró a tal opción del GPS
- escribo la calle San Lorenzo al 700
- el resultado esperado es que me muestre el punto en el mapa en el lugar correcto de nueva córdoba que corresponde

Otro caso de prueba, probar con una calle que no exista, precondición, la calle Chañar es una calle que no existe y no está cargada en el sistema, pasos a seguir:

- calle Chañar al 1400
- busco y el sistema tiene que dar como resultado un mensaje que diga "no se puede localizar la dirección ingresada"



Objetivo: la característica que yo quiero probar es la búsqueda de calles, en un caso de una calle y altura que exista y en otro caso una calle y altura que no exista.

## Casos de prueba

*“Los bugs se esconden en las esquinas y se congregan en los límites...” Boris Beizer*

Objetivo: descubrir errores

Criterio: en forma completa

Restricción: con el mínimo de esfuerzo y tiempo

Como el testing exhaustivo no es posible, lo que íbamos a intentar es probar lo que más podamos del sistema con el menor esfuerzo posible. Hay que pensar cómo definir y ejecutar la menor cantidad de casos de pruebas, pero que aseguren el máximo nivel de cobertura. ¿Como hago para realizar un caso de prueba que me permita cubrir varios aspectos de la funcionalidad?

En este contexto, una de las premisas que tiene que ver con plantear los casos de prueba es que normalmente los bugs o errores se concentran en los lugares límites, cuando estamos trabajando con rangos, entonces en este ejemplo de la búsqueda por calle altura, estaria bueno hacer una prueba que pruebe al 790 y al 791, porque el 791 no tiene que existir si la calle San Lorenzo va del número 0 al 790. De esta forma, me estoy asegurando de que esa prueba está tratando de contemplar la mayor cantidad de universos posibles.. Estoy suponiendo que cuando me voy al límite, como el 790, el 0 ó el 1, de esta manera tengo la total seguridad de que si el 790 anda el 500 va a funcionar y el 700 va a funcionar. De esta manera estoy diciendo que elijamos de estos casos de prueba y de estas variables, aquellos lugares donde probablemente hay más posibilidades de tener errores, si ahí funciona para el resto del conjunto también va a funcionar.

Hay una técnica que se llama prueba de caja negra por valores límites, que explora puntualmente esto.

## Derivación de Casos de Prueba



Hay distintas formas de enunciar los casos de prueba, si nosotros tenemos requerimientos planteados o User story, podemos derivar los casos de prueba desde ahí. Aclaración: si tenemos User Stories, **los casos de prueba no son las pruebas de usuario** que

escribieron en la User. Las pruebas de usuario tiene que ver con la prueba de aceptación que vamos a hacer en la review, hablamos de prueba y de criterios de aceptación que vamos a hacer en la review, está claramente vinculado con “¿hasta donde testeo?”.

Los casos de pruebas de los que hablamos acá no están en la User, son casos de prueba que son del lado del equipo para adentro. Las pruebas y los criterios de aceptación nos van a ayudar a definir los casos de prueba, pero los casos de prueba son otros.

Lo que se escribe en la User es lo que espero se espera como condición para saber si la User va a estar aceptada o no, son las pruebas de aceptación.

Las pruebas de usuario son aquellas que se hacen en ese último nivel, el de aceptación.

Estos casos de prueba que estamos planteando acá son los otros niveles, el unitario, de integración y de sistema.

Si no tenemos las User y tenemos casos de uso, podemos usar los casos de uso. Hay alguna forma que se conoce como **diseño conducido por casos de prueba o especificar requerimientos conducidos por casos de prueba**, son técnicas en las que uno en vez de escribir los requerimientos y después hacer los casos de prueba, escribe los requerimientos con forma de caso de prueba.

Si no tengo ninguna de esas cosas, empiezo a buscar, acá habla de la derivación del código, está la forma de construir software con TDD. Si tampoco tengo esas cosas, tendré que hacerlas con la información de relevamiento o desde los documentos del cliente.

Cuanto mejor se tengan especificados los requerimientos, más fácil van a salir los casos de prueba. Si los requerimientos son ambiguos cuando me siento a hacer los casos de prueba me voy a poner a especificar los requerimientos, con forma de casos de prueba.

## Conclusiones sobre la Generación de casos

- Ninguna técnica es completa  
Técnica de caja blanca, caja negra y combinarlas entre sí, ninguna técnica es completa y ninguna técnica nos permite abarcar todos los casos de prueba
- Las técnicas atacan distintos problemas  
Busca utilizar distintas técnicas complementarias entre sí, para tratar de abarcar la prueba de la mayor cantidad de SW posible con la ejecución de la menor cantidad de casos de prueba posible. ¿Cómo lo vamos a lograr? usando estas técnicas
- Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una  
Hay distintas técnicas que nos van a ayudar a enunciar el caso de prueba de tal manera que ese caso de prueba a mayor cantidad de escenarios posibles. Todas las técnicas tienen ventajas y desventajas y apuntan a cuestiones diferentes, tanto caja blanca como caja negra tiene sus ventajas, la técnica es combinarlas para obtener las ventajas de cada una. Tratar abarcar con el menor esfuerzo posible la mayor cantidad de casos de prueba estipulados, ese es el objetivo y el uso de esas técnicas nos ayudan a obtener eso. Estas técnicas nos ayudan a armar los casos de

prueba de tal manera que abarquen la mayor cantidad de funcionalidades cubiertas posibles

- Depende del código de testear
- Sin requerimientos todo es mucho más difícil  
Si no tenemos requerimiento especificados y definidos, cuando lleguemos al testing va a ser mucho más difícil porque nos tenemos que sentar en ese momento a pensar los requerimientos, si los requerimientos estan bien especificados, se simplifican de manera exponencial
- Tener en cuenta la conjetura de defectos
- Ser sistemáticos y documentar las suposiciones sobre el comportamiento o el modelo de fallas

### **Concepto: Ciclo de Test**

*Un ciclo de pruebas abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una misma versión del sistema a probar*

Supóngase que tenemos un conjunto de casos de pruebas con distintas técnicas, tenemos 10 casos y los ejecutamos. A veces vamos a tener los resultados esperados y a veces vamos a obtener defectos, estos defectos van a tener distinto grado de severidad, bloqueante, crítica, menor, etcétera.

Si nosotros terminamos de ejecutar todos los casos de pruebas y encontramos 20 defectos bloqueantes, vamos a tener que corregir esos defectos y el punto siguiente es volver a probar, para ver si esos defectos se corrigieron o no se corrigieron. Esa nueva vuelta de prueba, volver a ejecutar los casos de prueba, es un nuevo ciclo de test, es decir, cada vez que se ejecutan los nuevos casos de prueba que tengo definido, a una misma versión del sistema, es un ciclo de test. Termino esa ejecución, corrijo los errores y vuelvo a ejecutar, es otro ciclo de prueba.

Cada ciclo de test es la ejecución de la totalidad de los casos de prueba, aplicados a una versión, siempre es la misma versión.

¿Cuántos ciclos de prueba se ejecutan? Depende de los resultados de la ejecución de los casos de prueba. Si dijimos que el criterio de aceptación es que vamos a ejecutar casos de prueba hasta que no tengamos más defectos bloqueantes, entonces vamos a tener que ejecutar tantos ciclos de test, hasta que encontremos un ciclo del test en donde se hayan ejecutado todos los casos de prueba y no haya dado un resultado con un defecto bloqueante.

El ciclo de test va de la mano con la ejecución de la totalidad de los casos de prueba

### **Concepto: Regresión**

*Al concluir un ciclo de prueba, y reemplazar la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión, a fin de prevenir la introducción de nuevos defectos al intentar solucionar los detectados*

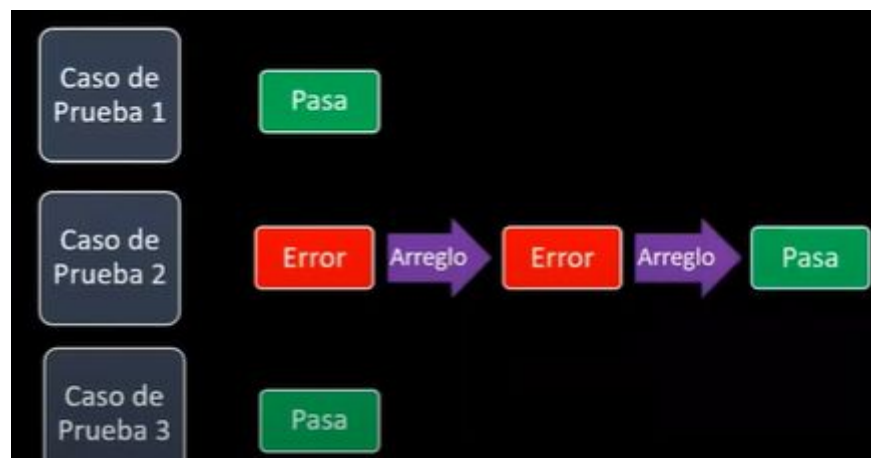
Uno de los aspectos que está directamente relacionado con el ciclo de prueba, es el concepto de regresión

**Pregunta:** ¿La ejecución del testing incluye la corrección de los defectos que encontrábamos? Si, lo encontramos en el concepto de ciclo de prueba.

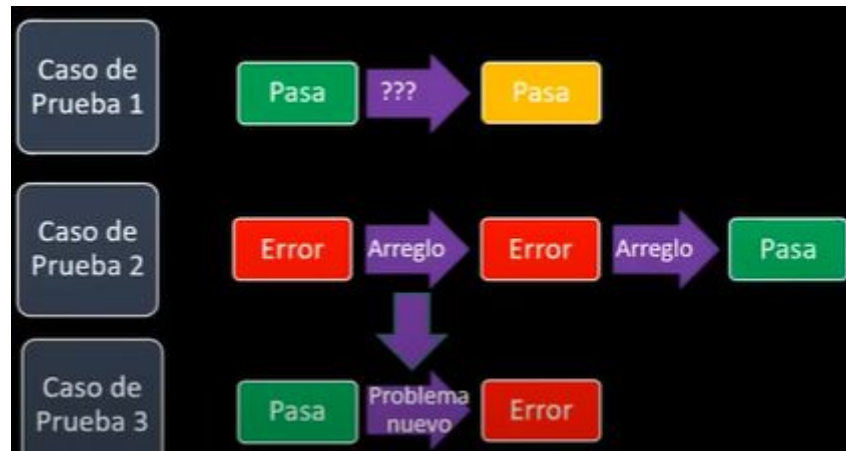
Ahora bien, la regresión lo que nos dice es que cuando terminamos un ciclo de prueba, y reemplazamos la versión habiendo corregido todos los defectos, lo que hay que hacer es volver a verificar todos los casos de prueba de manera completa, no simplemente en los que encontramos defectos. Hay que hacer esto, porque cuando corregimos un defecto, normalmente se introducen defectos nuevos y si solo nos limitamos a probar los casos de prueba en los que habíamos detectado los defectos para ver si se corrigieron, se nos pueden pasar por alto algunos defectos nuevos

### Ejemplo

Casos de prueba sin regresión

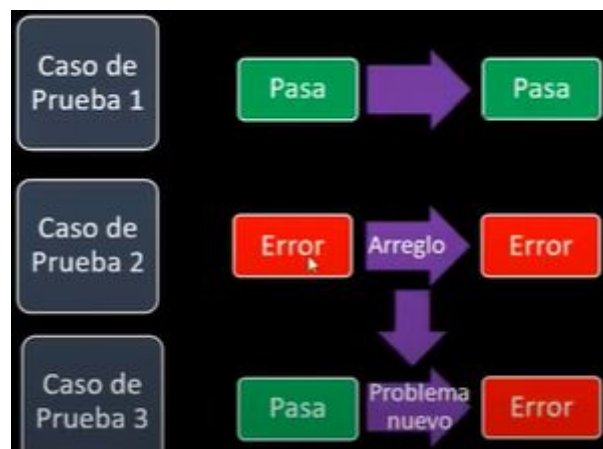


El caso de prueba 1 lo ejecutamos, anda todo bien, el caso de prueba 3 también, pero ejecutamos el caso de prueba 2 y encontramos defectos, los defectos se corrigen, volvemos a ejecutar y vuelven a salir defectos, volvemos a corregir y finalmente el caso de prueba se ejecuta de manera exitosa. Ahora bien, si nosotros nos paramos en ese lugar, ¿Qué es lo que pasa en esa ejecución?, nosotros solo nos fijamos en que pasa con el caso de prueba 2, que es donde habíamos encontrado el defecto. El tema es, ¿Qué pasa con el caso de prueba 1 y el caso de prueba 3? Aca estamos asumiendo que el caso de prueba 1 y el 3 no han tenido modificaciones, pero en verdad, la vida misma nos enseña que cuando nosotros hacemos correcciones en uno de los casos de prueba introducimos errores en otro de los casos de pruebas que están relacionados

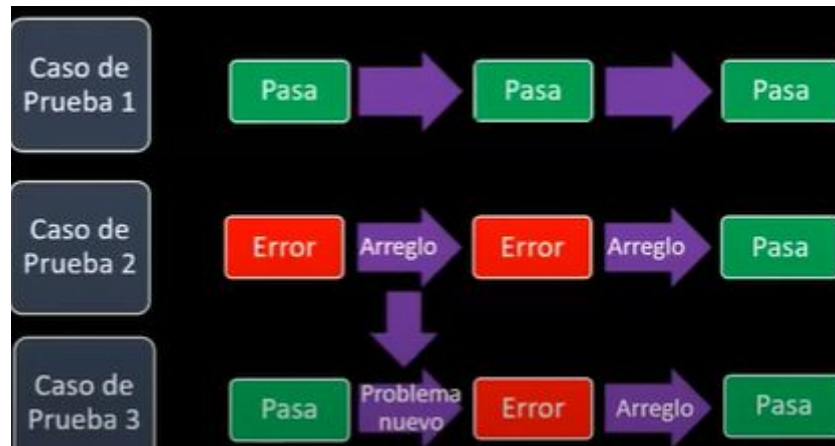


¿Qué pasa si no hacemos el test de regresión? ¿ Si no ejercitamos de forma completa todos los casos de prueba? Posiblemente introducimos defectos que nosotros pensamos que no existen porque en el ciclo anterior no estaban, pero en realidad si estan pasando producto de los cambios que hice para corregir los defectos que había encontrado

Casos de prueba con regresión



Plantea que cuando yo hago la regresión, pruebo todo de nuevo, fíjense aca, vuelvo a probar el caso de prueba 1, 2 y 3, y en el segundo caso de prueba tengo un defecto en el caso de prueba 2 y en el 3 ahora, entonces al hacer la corrección voy a focalizar la corrección de los defectos que aun tengo, y voy a ejecutar el nuevo caso de prueba



Después de que verifique que en ninguno de los tres hay defectos o que en el criterio de aceptación los defectos menores no se consideran determinantes para saber que sigo o no sigo, considero cerrado ese ciclo de prueba

El punto está en saber identificar aquellos defectos que se introducen producto de la corrección de los anteriores

### Concepto: Smoke Test

Smoke Test:

- Primera corrida de los tests de sistema que provee cierto aseguramiento de que el software que está siendo probado no provoca una falla catastrófica.

Hacer una primera corrida del sistema en términos generales, sin entrar en detalles para ver si no hay alguna falla catastrófica, Para no abocarme en la ejecución detallada de los casos de prueba cuando hay una ejecución mas grosera que se me está pasando en frente.

### Tipos de prueba

- Testing funcional

Las pruebas se basan en funciones y características(descripción en los documentos o entendidas por los testers) y su interoperabilidad con sistemas específicos

- Basado en requerimientos
- Basado en los procesos de negocios

- Testing No funcional

Es la prueba de “como” funciona el sistema

NO HAY QUE OLVIDARLAS!!! Los requerimientos no funcionales son tan importantes como los funcionales, a veces los casos de prueba solo en los requerimientos funcionales y eso es un error

- Performance Testing
- Performance de carga
- Performance de Stress
- Pruebas de usabilidad

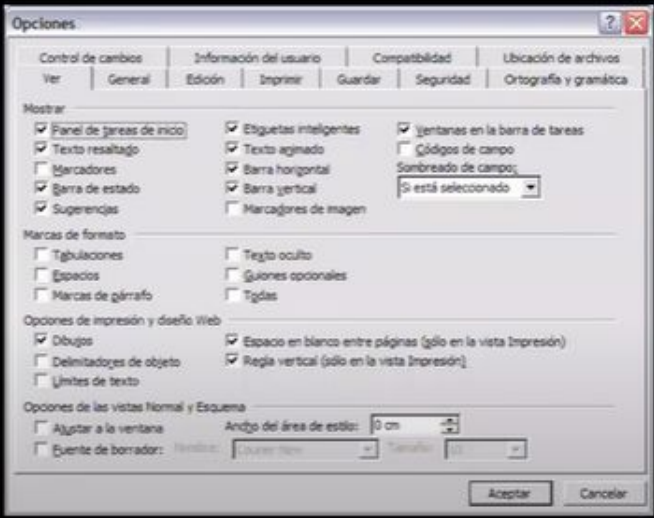
- Pruebas de mantenimiento
- Pruebas de fiabilidad
- Pruebas de portabilidad

El ambiente tiene que ser lo mas parecido al entorno de producción, porque aca estamos probando la carga, el stress, el performan. Esto se puede probar si tenemos un entorno parecido al entorno de real, sino esto es muy difícil de probar

- Pruebas de interfaces de usuario(prueba no funcional)

**Usuario en control**  
+  
**Muchas combinaciones**  
=  
**Más pruebas**

- Funciones de negocios
- **Interfaces de usuarios**
- Performance
- Carga
- Estrés
- Volumen
- Configuración
- Instalación



**Las GUIs,  
son mucho  
más  
complejas  
que las  
interfaces  
basadas en  
caracteres**

Normalmente las interfaces de usuario son cada vez mas complejas,ahora con la complejidad del usuario,estas pruebas son mas especificas y dedicadas a está area

- Prueba de performance(prueba no funcional)

**Prueba de performance**

- Tiempo de respuesta
- Concurrencia



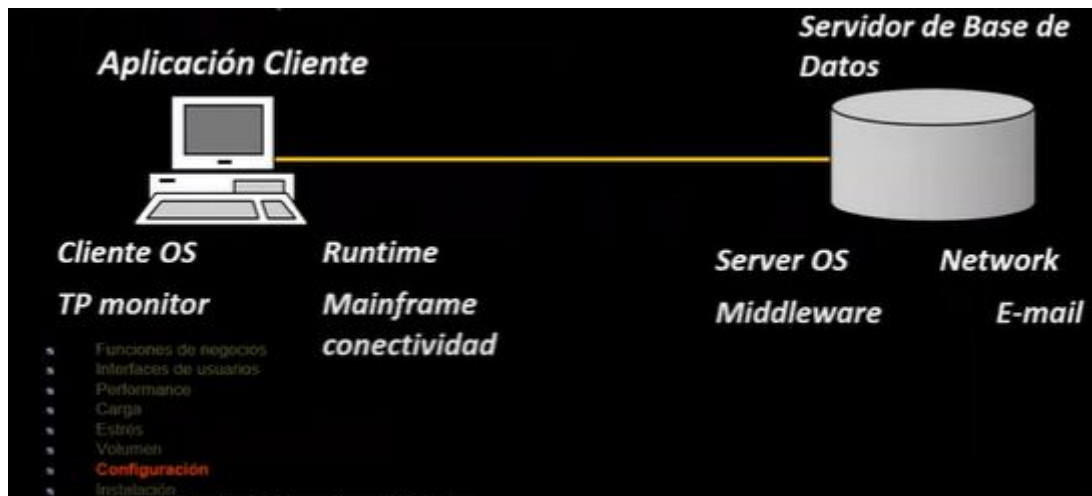
- Funciones de negocios
- Interfaces de usuarios
- **Performance**
- Carga
- Estrés
- Volumen
- Configuración
- Instalación

- Prueba de configuración (prueba no funcional)

Compatibilidad con X video drive

Conexiones de red

Software de terceras partes



Hacer la prueba de mi entorno con una determinada configuración de cada uno de los componentes. Si tengo una aplicación del tipo cliente/servidor con una determinada configuración del cliente, de la aplicación cliente servidor, incluso de lo que tiene que ver con las redes

## El Testing y el Ciclo de Vida

### Verificacion y validacion

Verificación - ¿Estamos construyendo el sistema correctamente?

Validación - ¿Estamos construyendo el sistema correcto?

El testing nos permite cubrir una parte, pero es la única forma, también están las revisiones técnicas

### El Testing en el ciclo de vida del software

Objetivo de involucrar las actividades de Testing de manera temprana:

- Dar visibilidad de manera temprana al equipo, de cómo se va a probar el producto
- Ayuda a ver si los requerimientos están correctamente especificados, tanto funcional como no funcional
- Disminuir los costos de correcciones de defectos

### Conclusión



Cuando hablamos del testing y del ciclo de vida, nosotros sabemos que tenemos distintos tipos de ciclos de vida, el ciclo de vida iterativos e incremental o el ciclo de vida en cascada. En la bibliografía se tiende a hacer una igualación entre el ciclo de vida en cascada y los procesos definidos.

Dependiendo del ciclo de vida, el testing se inserta de una manera distinta, dependiendo de cómo se plantea el ciclo de vida es como se plantea el testing. Es importante utilizar esos ciclos de vida en donde el testing se ejecuta lo más temprano posible

Pregunta: ¿Cuál es la diferencia entre running tester features y velocidad?

rtf mide cantidades de features que se van entregando(features que fueron probadas y están como software funcionando), aprobadas por el PO, pero es una métrica incremental, sprint a sprint. Nos muestra como va creciendo el SW a medida que pasa el tiempo, se ha dejado de usar, aca no se puede ver el tamaño de la features. Sirve para ver como avanza el proyecto y si está creciendo o no

La velocidad se mide en Story Points quemados en el sprint, esto no tiene que ver con la cantidad de features, sino con la estimación que tenia cada una de esas features.

En el rtf todas las features se cuentan como uno, en la velocidad se tiene en cuenta la estimación. Además, el rtf no se compara sprint a sprint, sino que es una métrica que se acumula, en cambio la velocidad se comprara sprint por sprint, buscando estabiliza esa métrica entre sprint quemados y aceptados para ayudarnos a establecer la capacidad y poder ver como el equipo va trabajando de una manera sincronizada, asegurarse de tener resultados similares en cada uno de los sprint.

## Práctico

Parte 1 – <https://www.youtube.com/watch?v=Zuc1VNO-wB8> ✓

Parte 2 – <https://www.youtube.com/watch?v=p78VFLY5wWE>

### *Recordatorio teórico*

¿Qué es el Testing?

*Proceso destructivo cuyo objetivo es encontrar defectos, es decir, asumir que una porción de sw por más trivial que sea puede y va a tener defectos y nuestro objetivo es encontrarlos para posteriormente resolverlos y contribuir a la calidad de un producto de software.*

¿Cuanto testing es suficiente? *Importante para la práctica.*

Depende de los criterios. Para un sistema o producto de sw no trivial, para poder probar toda la combinación de valores posibles que necesitamos para ejecutar funcionalidad llevaría un tiempo que no vamos a poder cubrir con los ciclos de pruebas.

Ya que los sw no triviales tienen funcionalidad mucho más complejas que ingresar campos, por ejemplo colaboración mediante apis, gps, antenas, etc, por lo que es imposible realizar la prueba de todas las combinaciones posibles.

¿Cómo hacemos para saber dónde cortar?

Hay distintos criterios:

- *good enough*: encontramos defectos y vamos resolviendo, existe una tendencia a disminuir defectos encontrados. Por lo que se decide cortar cuando la cantidad de defectos encontrados llegue a cierto umbral.
- *económico*: cuando se haya consumido el presupuesto destinado a ejecutar pruebas.

Las metodologías ágiles hacen énfasis en la **automatización** de las pruebas para economizar los recursos disponibles y repetir pruebas, hacer pruebas de regresión (controlar que los defectos que arreglamos no hayan producido otros defectos).

ERROR vs DEFECTO: (pregunta de final)

Ambos son fallas en el producto de software.

Estas fallas son errores o defectos dependiendo del *momento* donde se producen. Cuando se encuentra una falla en la misma etapa donde se originó es un ERROR, ejemplo en implementación, cuando falla trasciende de la etapa de dónde se originó es un DEFECTO, ejemplo en producción, en las pruebas alfa o el entorno de aceptación.

## **El testing busca encontrar DEFECTOS**

¿Por qué no busca encontrar errores? No es el objetivo del testing como disciplina, aunque hay *un nivel de testing* donde *podemos* encontrar errores.

El testing se ejecuta como disciplina en una instancia posterior a la implementación y brinda una retroalimentación con la implementación.

En el testing unitario se encuentran *errores* ya que trabaja sobre la etapa de implementación, pero en si el objetivo del testing unitario no es ese. El testing unitario puede encontrar errores pero no busca encontrarlos.

Si ese error durante las pruebas unitarias no es encontrado y trasciende hablamos de un *defecto*. (Dejamos pasar la falla más allá de la implementación)

- El error es más barato. Se prefiere encontrar errores, para identificarlos antes que sea muy caro corregirlo. Encontrar un error en la producción es mucho más caro que arreglarlo en la implementación.
- Caro significa -> A nivel de \$, relación con el cliente, esfuerzo, etc.
- El testing busca estas fallas antes que vayamos a producción.

Siempre vamos a buscar encontrar estas fallas lo antes posible. Vamos a buscar evitar que llegue al entrono productivo.

### PROCESO DEL TESTING:



Nosotros trabajamos en el *ANÁLISIS Y DISEÑO* de los casos de pruebas que son el artefacto por excelencia en el testing como disciplina y que sirven como guía en la ejecución de pruebas para luego realizar una retroalimentación en base a los reportes y evaluación de los resultados.

**Pregunta:** *¿En la integración continua se encuentran defectos, ya que se corren tests en esa etapa?*

Encontramos defectos porque ya pasaron de implementación. Vos terminaste de implementar esa funcionalidad y hiciste CI (porque es parte del criterio de done incluirlo a la rama master) y ahí detectas defectos sobre funcionalidades que ya estaban implementadas.

- En las prácticas continuas podemos automatizar las pruebas de sistema y pruebas de aceptación. Siempre soportado con las herramientas.

### NIVELES DE TESTING

- **Testing unitario:** cuyo objetivo, a diferencia de la disciplina, no es encontrar defectos (inicialmente por cuando se desarrolla, su momento es en la implementación) ni errores. Busca proveer un mecanismo de verificación de la comprensión de la funcionalidad que estoy implementando (Help check de que lo que estoy haciendo está bien). Escribo los casos de prueba unitaria que me permiten garantizar que la funcionalidad que estoy implementando cumplen con el requerimiento dado (Verificación de lo que estoy trabajando). Estos no deben fallar, por el hecho que están hechos para garantizar que eso está bien en el momento que se está implementando. Si aparece algo, es un error porque todavía estoy a tiempo de corregirlo antes de pasar a la siguiente etapa. Lo lleva adelante el desarrollador. El equipo de testing no soluciona errores, ejecutan diseño y reporta los

casos de prueba al equipo de desarrollador. En cambio, estos tests lo ejecuta el desarrollador, el mismo es quien corrige y le da luz verde a las pruebas

- **Testing de integración:** Buscamos entrar defectos, está orientado a probar las interfaces, a nivel de fronteras entre componentes de mi producto de software (componentes de todos los niveles). Buscamos probar las interacciones entre componentes. Hay varias formas para realizar este testing: bottom up (desde los componentes más chicos -clases- a los más grandes -sistemas o subsistemas-), top down (desde componentes más grandes a los más pequeños), sandwich (ir alternando), nos dan un orden de ejecución de los testing de integración. Lo realiza el tester.
- **Testing de Sistema:** probar una funcionalidad en su totalidad, donde su funcionalidad ya está implementada el tester ejecuta un camino para encontrar si el resultado de la ejecución de esa funcionalidad es el esperado y así controlar defectos. Lo realiza el tester.
- **Testing de aceptación:** tarea del Usuario, cliente. Validan que lo entregado corresponde con sus requerimientos manifiestos como con sus expectativas. Validamos que lo entregado corresponde con el requerimiento planteado. No se deberían encontrar defectos, no es su objetivo. Es una forma de generar una aproximación.

Como no podemos probar todo ya que existen infinitas combinaciones de valores a configurar en nuestra funcionalidad, por lo que surge la necesidad de plantear criterios *económicos* para el testing, es decir poder maximizar la cantidad de defectos encontrados minimizando el esfuerzo requerido para hacerlo, para ello se utilizan **ESTRATEGIAS** para el diseño de casos de prueba.

**Casos de prueba:** artefactos por excelencia del testing que contiene condiciones y pasos a ejecutar para garantizar que el resultado esperado sea o no igual que el resultado obtenido y así buscar defectos.

Vamos a buscar diseñar la menor cantidad de casos de prueba para maximizar la cantidad de defectos encontrados. (Economizar)

- ★ **Estrategia de Caja Blanca:** donde vemos el detalle de la implementación de la funcionalidad y disponemos el código, podemos diseñar el caso de prueba para garantizar la COBERTURA. Ej: cubrir todos los if, else, ramas de ejecución, etc.
- ★ **Estrategia de Caja Negra:** no conocemos la estructura interna, solo podemos analizar las entradas y salidas, elegimos los valores para ejecutar, y obtenemos las salidas obtenidas con las que esperaba obtener.

Lo ideal es hacer una combinación de estrategia, esto lo decide cada equipo.

Estrategia → Implementación de una estrategia en particular

¿Y para qué los usamos? → Para maximizar la cantidad de defectos encontrados)

### **Método de las estrategias o implementación**

- Caja negra:
  - Basados en especificaciones:

- **Partición de equivalencia o clases de equivalencia:** analiza primero cuales son las **condiciones externas** involucradas en el desarrollo de la funcionalidad, pueden ser entradas y salidas, una vez analizadas para cada condición externa defino los subconjuntos de valores posibles que pueden tomar cada una que producen un resultado equivalente.

Ejemplo: ingreso a un sitio web de bebidas alcohólicas y me pregunta la edad para permitirme ingresar, mi condición externa de entrada es: *edad*, la que se puede dividir en subconjuntos posibles que cada uno me devolverá un resultado para la funcionalidad. En este caso son dos equivalencias: la primera, numero enteros mayores a 0 y menores a 18, la segunda mayores o igual a 18 y menores de 100. Condiciones externas: edad (entrada), y el ingreso (salida).

- **Análisis de valores límites:**
  - Una particularidad de la partición de equivalencias
- Basados en experiencia
  - **Adivinanza de defecto**
  - **Testing exploratorio**

## Caja Negra: Partición de Equivalencias

Dos Pasos

1. Identificar las clases de equivalencia (Válidas y No Válidas)
  - a. Rango de valores continuos
  - b. Valores discretos
  - c. Selección simple
  - d. Selección múltiple

Ejemplo:

Subconjuntos de valores de edad que pueden llegar a obtener un resultado equivalente

- Números enteros mayores o iguales a 18 y menores a 100
- Números enteros mayores o iguales 0, y menores a 18
- Números mayores a 100
- Números menores a 0
- Valor no numérico (letras, símbolo, emojis) → clase de equivalencia no válida
- No ingresa valor → clase de equivalencia no válida
- Número no entero → clase de equivalencia no válida

Y va a haber diferentes resultados. La división en subconjuntos se realiza en base a los **resultados** posibles que se van a obtener. (En este casos son los mensajes de error). Cualquier valor que tome de ese subconjunto va a producir un resultado equivalente. Esta es la esencia de la partición de equivalencia.

*(Pregunta de final)*

**Clase/partición de equivalencia:** subconjunto de valores que puede tomar una condición externa, para el cual si yo tomo cualquier valor o miembro de ese subconjunto, el resultado de la ejecución de la funcionalidad es **equivalente** (no igual, equivalente).

Hay que buscar los subconjuntos que, unidos, conforman el universo de posibilidades o valores para una condición externa dada.

Una vez que identifican las clases de equivalencia, van a poder encontrarse con diferentes casos:

- Rango de valores continuos
- Valores discretos
- Selección simple
- Selección múltiple

Si tienen que seleccionar una ciudad de un listado, van a tener una clase de equivalencia que sea de selección simple, es decir, el conjunto de equivalencias va a ser una ciudad dentro del listado cordoba, villa allende, rio ceballos. Pueden ser números, archivos, coordenadas del gps, cualquier cosa.

## 2. Armar los casos de prueba

Una vez que se identificaron todas las clases de equivalencia y las condiciones externas, se van a armar los casos de prueba.

Aca es donde el metodo tiene su resultado. Los métodos sirven para poder diseñar casos de prueba.

Para armar un caso de prueba, voy a tomar de cada una de las condiciones externas, una clase de equivalencia particular y voy a elegir un valor representativo de la misma para poder conformar mi caso de prueba.

**Caso de Prueba:** Receta. Conjunto de pasos ordenados que debo seguir para ejecutar la funcionalidad con una especificación de cuáles van a ser los valores que voy a ingresar. Entonces un CP te va a decir voy a ejecutar o tratar de ingresar al sitio web de la bebida alcohólica con una edad que supere la mayoría de edad.

- Paso 1: ingresar en el campo de edad 18
- Paso 2: apretar el botón ingresar
- Resultado esperado: que me muestre un mensaje de bienvenido y que cuando lo ejecute pueda ver si coincide o no

El CP debe especificar valores que son miembros de alguna clase de equivalencia de esa condición externa.

¿Para que me sirven las Clases de Equivalencia? Para que cuando diseñe esos casos de prueba y elija valores para ingresar a las diferentes condiciones externas de entrada, voy a tomar solo 1 de cada clase de equivalencia, porque cualquiera que tome va a producir un

resultado equivalente, entonces no me hace falta elegir más de uno. (Sino sería redundante probar con más valores, impacta en la economía del caso de prueba)

Resultado equivalente entonces al fin y al cabo se traduce en una peor economía a la hora de crear esos casos de prueba.

La idea es que ustedes ahora aprendan el método para que diseñen caso de prueba re económico y muy eficientes.

No hay ninguna verificación entonces por encima de eso? Yo supongo que las clases y los intervalos están bien definidos y pruebo un solo valor. No se puede probar un conjunto de números o algo así?

No, la idea en este caso particular es que el caso de prueba tenga un representante de clase equivalencia porque vos ya por definición te dice que cualquier otro valor de la clase de equivalencia producen resultados equivalentes, entonces no necesitas ejecutarlo con otros valores. Esa es la razón de ser de la partición equivalencia.

Por ejemplo si lo haces mal y vos podría identificar los dos clases de equivalencia con respecto a las edades válidas, podría decir que desde 18 a 30 y de 30 a 100 y el resultado es que ingrese al sitio web. Y si, lo podría hacer pero en realidad vas a hacer dos pasos de prueba para poder cubrir las clases de equivalencia que ingresa 24 y 35, cuando no te hace falta hacerlo entonces perdés tiempo, perder recursos y no cumplir con la economía del diseño de casas prueba.

El testing es una disciplina y es un rol el tester que se desarrolla y que consigue su expertise y esto es el inicio de todo lo que este fin que es enorme como disciplina.

### **¿Cómo vamos a ver esto representado en el práctico?**

Primero vamos a identificar las CE que participan en esa funcionalidad (tanto de entrada como de salida)

Ejemplo: edad, ciudad, etc (Según la funcionalidad dada).

Luego separamos los subconjuntos de clases de equivalencia válidas contra clases inválidas.

Ej para edad:

- Num ent entre 18 y 100 ambos incluidos → válidas
- Num enteros menores a 18 y mayor a 0 → inválidas

Condición externa	Clases de equivalencia válidas		Clases de equivalencia inválidas	
<i>edad</i>		<i>Numeroos enteros entre 18 y 100 ambos incluidos</i>	<i>Numeeroos enteros menores a 18 y mayor a 0</i>	

hasta Min 11:42



## Parte práctica parte 2 - Desde el minuto 15:58

Id del Caso de Prueba	Prioridad (Alta, Media, Baja)	Nombre del Caso de Prueba	Precondiciones	Pasos	Resultado esperado
			El usuario "Juan" esta logueado con permisos de administrador. La fecha actual es 15/5/2020	<ol style="list-style-type: none"><li>1. El cliente ingresa a la opción "Ingresar"</li><li>2. El cliente ingresa un valor mayor a 18 en el campo de "edad"</li><li>3. El cliente selecciona la opción ingresar</li></ol>	<ol style="list-style-type: none"><li>1. Se muestra el mensaje "Bienvenido al sitio web"</li></ol>

Por ejemplo: si nuestra funcionalidad es ingresar al sitio web, el nombre del caso de prueba va a ser ingresar con una edad menor a 18 años, o ingresar con una edad de adulto. O uno de probabilidad baja sea ingresar al sitio web sin ingresar la edad o con una edad no numérica. Tiene que ser un nombre que represente el escenario por el cual yo voy a estar ejecutando este caso de prueba.

Después, los pasos va a ser un conjunto de operaciones ordenadas y numeradas en las cuales el formato de redacción es:

- 1) El ROL ingresa a la opción "Ingresar" (en este caso puede ser el cliente o visitante)
- 2) El cliente ingresa "18" en el campo de "edad" y después el cliente selecciona la opción ingresar.

Entonces es un conjunto ordenados de pasos totalmente claros sin ambigüedades que deben ejecutar nuestros testers sobre la funcionalidad para conseguir un resultado esperado. El mismo, es "Se muestra el mensaje Bienvenidos al sitio web".

Después las precondiciones es todo el conjunto de valores o de características que tienen que tener mi contexto para que yo pueda llevar adelante este caso de prueba en particular. ¿A qué me refiero? Por ejemplo, cuando se requiere que el usuario este logueado, o que tenga ciertos permisos para ejecutar cierta funcionalidad. O casos de prueba que dependen del día en que se ejecutan.

**Pregunta: ¿Y cosas muy finas como por ej. que tenga datos móviles, o el GPS activado, sería una precondición, o no deberíamos tenerlo en cuenta?**

En los casos del caso de prueba, salvo que la funcionalidad dependa de eso, no es necesario especificarlos. Si tuviera que buscar un taxi, no solamente le vas a poner que tiene que tener el GPS encendido sino que las coordenadas actuales son 31 64 27 por ejemplo. Entonces vos después vas a poder reproducir exactamente el caso de prueba en ese escenario dado. Si el tester es un australiano que está trabajando con vos, y ejecuta el caso de prueba, si vos no especificas ese valor en una pre condición el resultado puede ser diferente.

Las precondiciones y los casos y los resultados esperados tienen que especificar si o si valores identificables para cada una de esas clases de equivalencia, tiene que ser un valor concreto. Si yo pongo por ejemplo "el usuario está logueado con permiso de administrador" está incompleto porque si después el tester ejecuta el caso de prueba con el usuario "Juan" quizá el usuario Juan tenga otros registros o historial o datos que no sea el usuario "Pedro" entonces eso no va a poder garantizar la reproducibilidad del caso de prueba. Si yo no pongo la fecha de hoy quizá el cálculo sea distinto. Por eso siempre tiene que tener un valor concreto. Si yo pongo en edad "el cliente ingresa un valor mayor a 18" esta mal, porque tiene que ser exactamente cuál es el valor que va a ingresar en ese campo. Lo mismo para el resultado. Si dice que muestra un mensaje, especificar cual.

**Pregunta: ¿Los resultados esperados siempre son de camino feliz?**

No. De ese escenario que estás ejecutando. Si vos ejecutas el escenario donde no ingresa la edad el resultado esperado va a ser "el sistema muestra el mensaje debe ingresar una edad". Casualmente el objetivo es que cuando el tester ejecute esos caso de prueba con esas precondiciones dadas pueda comparar el resultado obtenido contra el que vos decís que debe tener, qué es el resultado esperado. Si no hay match, no coinciden, hay defectos.

**Pregunta: Para la situación de "Llene todos los campos" ¿Hace falta un caso de prueba o no hace falta?**

En este caso en particular vos si tenes mas de un valor de entrada, vas a poder hacer un escenario completo que sea "no se ingresa ningún valor" o "no se ingresan los campos obligatorios". Entonces en tu caso de prueba vas a poner específicamente que si ingresó y el resultado esperado puede ser un solo mensaje que diga "debe completar todos los campos obligatorios" o según como esté definido el caso de prueba puede ser un conjunto de mensajes.

**Pregunta: Más allá de que haya un único mensaje o no, ¿Vamos a tener que hacer un caso de prueba por cada combinación posible?**

No necesariamente.

**Pregunta: ¿Sobre la clase equivalencia campo obligatorio y puedo tomar uno y listo?**

Osea no vas a tener nunca una clase equivalencia que se llame campos obligatorios. Vas a tener por ejemplo condición externa nombre, condición externa apellido, condición externa edad, y cada uno de esos va a tener una clase equivalencia inválida, que no ingresa valor, no ingresa valor, no ingresa valor. Entonces tu caso de prueba vas a usar esas tres condiciones externas y para estas tres condiciones terna vas a usar cada una, una clase de equivalencia, que va a ser el no ingresa valor. Entonces tu caso de prueba va a ser, por ejemplo, registrar cuenta de usuarios sin ingresar valores. Y vas a utilizar las tres condiciones externas, nombre, apellido y edad, con las tres clases de equivalencia inválidas de no ingresa valor en el mismo caso de prueba.

**Pregunta: ¿Y si en un caso de prueba tenemos varias salidas, osea varias pantallas?**

Lo podes especificar en el resultado esperado. El resultado es "el sistema muestra tal cosa" o "el sistema muestra tal otra" y puede mostrar varias cosas juntas. No hay ningún problema, pero tiene que quedar claro exactamente qué es lo que muestra.

**Pregunta:** ¿Cuando se hablaba de las precondiciones (retomando el caso que habíamos visto del bar) si yo digo que para entrar al bar tenés que tener ya sea una membresía o una reserva ¿Es una precondición o forma parte de la validación?

En ese caso en particular, si debes tenerla y está asociada a tu cuenta de usuario va a entrar como una **precondición**, porque no es una condición externa que puedas ponerle a la funcionalidad, si no que depende de que el usuario logueado tenga una reserva anteriormente. Si por ejemplo, tenemos que poner la edad y el número de reserva ahí es una **condición externa de entrada** porque es un valor que podemos ingresar a esa funcionalidad.

Precondición: El usuario Natalia se encuentra logueado y tiene una reserva para el día 19 de mayo a las 20:00 hs

### **Análisis de Valores Límites**

La variante del método de partición de equivalencias es el Análisis de Valores Límites.

Para poder escribir casos de prueba en base al método de partición de equivalencias tomábamos un valor cualquiera de la clase equivalencias ya que producía un resultado equivalente. El Análisis de Valores Límites nos dice que la mayor cantidad de los defectos se encuentra en los extremos de los intervalos.

28:20