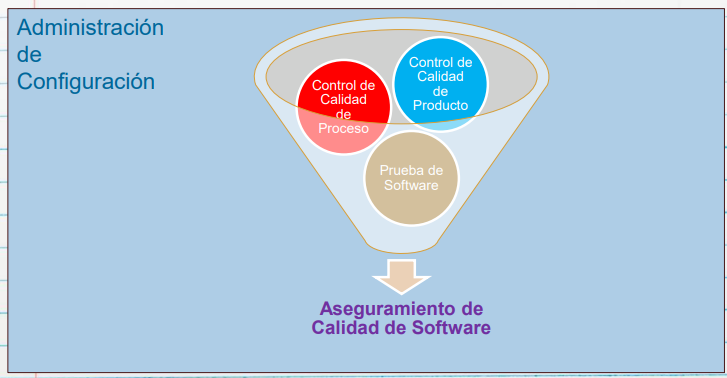
**Prueba de Software o Testing**

Para entrar en contexto, habíamos dicho que el Aseguramiento de Calidad de Software consta de 3 actividades: el control de calidad del proceso, el control de calidad del producto y la prueba de software. Las primeras dos, se realizan mientras el software se construye, mientras que la **Prueba de Software o Testing** se realiza cuando el producto ya está construido, y es por eso decimos que el Testing “”llega tarde”. Como vemos, las anteriores actividades se realizan durante el proceso donde el producto se construye, en el proyecto, debido a que la calidad NO se agrega al final, sino que hay que garantizar, controlarla y evaluarla mientras construimos el producto, porque cada decisión que tomamos en un proyecto afecta a la calidad del producto.

Además, controlamos la calidad mientras se construye porque un defecto, a mayor edad (con edad nos referimos al tiempo transcurrido desde que existe hasta que lo detecto y lo corrijo) es más costoso cambiarlo y repararlo, ya que se traslada a otras partes del producto. Incluso si el cliente es quien lo detecta, es aún peor, porque tiene un costo intangible: la disconformidad del cliente.

**Concepto de testing**

El testing se utiliza para controlar la calidad, para poner en evidencia defectos, pero no hago aseguramiento de calidad sólo con testing porque no agrega calidad ni asegura que el producto no tenga errores. Esto es un error común en analistas funcionales que firman SQA (Quality Assurance) de un producto, pero esto no es correcto si solo hago testing.

Podemos definir al testing entonces como el **proceso destructivo** de tratar de **encontrar defectos cuya presencia se asume** en el código (es decir, SIEMPRE tendremos errores, porque es algo propio de la naturaleza humana, lo que implica que siempre necesitamos testing, y sabemos que los hay). Se debe ir con una actitud negativa para demostrar que algo es incorrecto (lo normal es que haya errores, el desafío es encontrarlos). Por lo tanto, un testing exitoso es aquel que encuentra defectos, y es una actividad muy costosa que, mundialmente, se lleva del 40% al 50% del costo de un software confiable, y este costo viene por el re-trabajo (cantidad de ciclos de prueba) y la cantidad de horas personas necesarias para hacerlo. Es por ello que no podemos tener una cobertura del 100%, no es posible probar todo, y esa cobertura se **acuerda** en las **pruebas de aceptación** (por lo tanto, hay una aceptación de ellas por parte del PO).

Lo ideal a la hora de realizar testing es que se encuentren errores, se corrijan y luego no se encuentren más, es decir, la cantidad de ciclos de prueba ideales son 2: la prueba sobre el desarrollo inicial o ciclo 0, y la segunda corrida sobre el software corregido, donde no encontramos más errores

La Administración de Configuración de Software (SCM) nos da el “paraguas” para hacer aseguramiento de calidad, ya que sin SCM no puedo hacerlo, ya que incluso hay partes o tareas de las actividades del aseguramiento de calidad que se encuentran dentro de una actividad del SCM: las **auditorías**. Además, dentro del aseguramiento de calidad incluyo el control de calidad del proceso. ¿Por qué esto es así?

Podemos decir que “la calidad del producto depende de la calidad del proceso que se usa para construirlo”, y es por eso que se definen también estándares de calidad del proceso, pero no hay estándares que acrediten calidad del producto, porque el proceso es la única “variable” que **puedo controlar.**

Algo tiene **calidad** si cumple con las necesidades y expectativas del cliente, y esto es visible en los requerimientos del producto, ya que el producto debe cumplir lo que se refleja en ellos, y es por ello que no existen estándares para calidad de producto, debido a que es **altamente personalizado**.

Es importante destacar que durante el testing se controla **código**, por lo tanto, no lo podemos hacer sin código, y no asegura ni la calidad en el software (calidad en la construcción) ni software de calidad (que haga lo que tiene que hacer)

**Diferencia entre error y defecto**

Por un lado, los **errores** se cometen al programar, y son detectados en la misma etapa en la que se producen, mientras que los **defectos** son aquellos errores que se trasladan a etapas posteriores siendo más costosos de corregir, y ambos pueden provocar o no **fallas**, es decir, un mal funcionamiento del sistema. Por ejemplo, un error de ortografía es un defecto que no produce una falla. En el testing, siempre buscamos encontrar **defectos**, porque como el testing siempre “llega tarde”, se realiza luego de la programación.

**Conceptos: defectos, severidad y prioridad**

Los defectos se pueden clasificar por **severidad** y por **prioridad**. La primera viene definida por la persona que prueba, mientras que la segunda viene del lado del cliente, es decir, en cuánto complica la operación diaria; y es verdad que ambas están relacionadas pero no necesariamente la relación es 1 a 1 (a más severidad, más prioridad y viceversa).

Con respecto a la severidad podría ser:

1. Bloqueante: el sistema no funciona
2. Crítico: da malos resultados o existe un mal funcionamiento, por lo que al final no puedo ejecutar o hacer lo que preciso, pero el sistema no se cae
3. Mayor: puedo hacer lo que quiero, pero el resultado es erróneo
4. Menor: puedo terminar el proceso pero se presenta una advertencia, aunque se puede continuar igualmente
5. Cosmético: son defectos de interfaz, ortografía, distribución de pantalla, formateo, etc.

Con respecto a la prioridad podría ser: Urgencia, Alta, Media y Baja

Esta es una forma de establecer las escalas, pero podría implementarlo de otra forma sin ningún problema, según mis necesidades. Muchas veces la severidad puede variar de empresa a empresa, por ejemplo, un defecto en los colores de la interfaz para una empresa de ciberseguridad es un defecto cosmético, quizá con prioridad baja, pero para una empresa de marketing seguramente es de prioridad alta.

**Niveles de prueba**

Existen 4 niveles de prueba., que determinan el foco de lo que voy a probar, probando desde lo más chico hasta lo más grande (al contrario que en el desarrollo, que voy desde lo más general como los requerimientos, a lo más particular como objetos o clases), es decir, parto de las pruebas de unidad que prueban lo más chico, y termino con las pruebas de aceptación de usuario. Todas deberían ser realizadas por el tester a excepción de las pruebas unitarias, ya que está integrada en el trabajo de los desarrolladores (en el Definition of Done). Esto es así porque uno mismo no debería probar lo que uno hace porque es más complicado detectar errores propios, ver la solución desde otra perspectiva que no sea la propia; excepto en las pruebas unitarias, y es de aquí que surge el TDD o Test Driven Development (Desarrollo dirigido por pruebas).

**Prueba unitaria:** En ellas pruebo la unidad mínima funcional, que en POO es una clase, mientras que en Paradigma Estructurado es un módulo. Es decir, está enfocada a los componentes más pequeños de software que se pueden probar y es realizada por el desarrollador durante la implementación, intentando proveer un mecanismo para verificar la funcionalidad de lo que se está implementando. En este caso sólo descubrimos errores, porque aún no se cambió de ambiente: se generó y se detectó en el ambiente de desarrollo.

**Prueba de integración o de interfaz:** las pruebas unitarias realizadas en el desarrollo ya me permiten saber si las unidades mínimas funcionan por separado, por lo tanto, en estas pruebas el objetivo es probar si funcionan en conjunto, si se comunican correctamente (es distinto a probar un sistema integrado), intentando encontrar defectos en la unión de los diferentes componentes. La integración debe ser incremental y puede ser realizada de arriba hacia abajo (top-down) o de abajo hacia arriba (bottom-up) El resultado será una **build** (sólo código y bases de datos, sin incluir documentación u otras cosas, a diferencia de una línea base). Esta build una vez que sea probada y cumpla con las pruebas (aceptado por el cliente) se transforma en un release (se pone en producción o se lanza al mercado). En teoría, debería ser llevada a cabo por el área de testing pero muchas veces en la práctica está integrada en el desarrollo. Los puntos claves de esta prueba son: conectar de a poco las partes más complejas y minimizar la necesidad de programas auxiliares

**Prueba de sistemas o prueba de versión:** pruebo “lo que hay” del producto, es decir, un incremento de producto. Esta tarea es llevada a cabo por el área de testing y NO es negociable, ya que no pueden probarlo los desarrolladores sino que debe hacerlo un tercero, caso contrario, buscarían demostrar que no hay errores, cuando lo que queremos hacer en realidad es encontrar los errores que presuponemos que existen. Este tipo de prueba se realiza para conocer la estabilidad de la aplicación antes de realizar una prueba en profundidad, cuando la misma ya está funcionando como un todo. Por lo tanto, el entorno de prueba debe ser lo más similar posible al de producción para reducir al mínimo los riesgos de incidentes debido al ambiente. Las funcionalidades tienen visibilidad para el cliente y puede ser encarado por QA. Aquí se incluyen los Smoke Test o los Sanity Test (donde se re-prueban los elementos clave).

**Pruebas de aceptación:** Es la prueba realizada por el usuario para ver si la aplicación se ajusta a sus necesidades, y se realiza tanto en ambiente de laboratorio como en ambientes reales. El objetivo es establecer confianza en el sistema, tanto de las características funcionales como no funcionales, por lo que encontrar defectos no es el foco principal. En la medida de lo posible el usuario o cliente debe estar presente, y nosotros también debemos estar presentes (testers), no solo le enviamos el sistema al usuario para que lo pruebe. El cliente debería tener sus pruebas de aceptación, sin embargo, en la práctica muchas veces se las facilita el equipo de desarrollo. En SCRUM se realiza durante el Sprint Review y son probadas por el Product Owner y los clientes que él invite. Mientras que en el PUD se realizaría durante el despliegue.

En el testing, existe un proceso sistemático y metodológico, ya que un defecto es aquel que se puede **reproducir** (se debe detallar el paso a paso para poder encontrar ese defecto y así poder corregirlo, y es por ello que utilizamos, por ejemplo, casos de prueba). En cambio, el testing ad hoc, es decir, “probar por probar” o “al voleo”, sin detallar un paso a paso, sólo probando sin ningún tipo de estructura ni orden **no es hacer testing**, debido a que no puedo reproducir el defecto y no aporta valor en ningún sentido, no permite corregir.

**Ambientes para la construcción del software**

En los distintos ambientes voy a encontrar todos los recursos de software y hardware necesarios para construir el producto. Tenemos 4 ambientes:

**Ambiente de desarrollo:**  en este ambiente se encuentran todos los recursos de HW y SW para poder desarrollar un producto. Es en este ambiente donde se realizan las pruebas unitarias, e incluso las pruebas de integración. No pueden acceder las personas de testing.

**Ambiente de prueba:** Es potestad de la gente que prueba, por lo que es un ambiente limpio donde no tienen acceso las personas encargadas del desarrollo. En este ambiente, se encuentra todo el SW y HW necesario para poder probar y testear el producto. Es muy importante la separación con el ambiente anterior, por un lado por lo que dijimos que uno mismo no debería probar lo que uno desarrolla, pero también porque es muy común que una cierta funcionalidad funcione correctamente en una PC de desarrollo, pero no en otras, lo cual hace necesario el cambio de ambiente.

**Ambiente de pre-producción:** es un ambiente que se utiliza para probar o testear cosas sin comprometer el ambiente de producción, por ejemplo, para probar cambios de software en una TELCO sin que el servicio esté caído durante la prueba ni esté afectado si se llegan a cometer errores. Sin embargo, tener un ambiente de preproducción es caro, por lo que muchas veces no existe.

**Ambiente de producción:** ambiente donde el producto construido es utilizado en la realidad, en la operación diaria por el cliente.

Estos últimos dos ambientes suelen estar en uno solo en muchos casos, pero en otros es vital la separación de ambos, por ejemplo, en un banco, donde un error al implementar un cambio en el ambiente de producción puede implicar pérdidas millonarias.

**Caso de prueba**

Un caso de prueba es una **secuencia de pasos** o conjunto de acciones a seguir para llegar a un objetivo. Y no sólo eso, si no que hay que sumarle también el **resultado esperado** y las **condiciones de seteo o precondiciones**, es decir, lo que tiene que estar presente o disponible para poder ejecutar la prueba, por lo que implica fijar una situación para poder ejecutar el testing.

Podemos decir que es un set de condiciones o variables bajo las cuales un tester determinará si un software está funcionando correctamente o no. Una buena definición de casos de prueba nos ayudará a **reproducir defectos.**

Es el **artefacto** más importante del testing, ya que es la salida referencial de los procesos de prueba. Estos se diseñan una vez y lo puedo repetir o ejecutar cuantas veces quiera (son reutilizables), es un proceso sistémico que me permite saber qué probé y qué no probé. Además, es importante diseñar la menor cantidad de casos de prueba para tener la mayor cobertura posible, para reducir el esfuerzo. Como no siempre tengo tiempo para ejecutar todos, tienen asignada una cierta **prioridad**.

Entonces, un caso de prueba busca descubrir errores (objetivo), en forma completa (criterio) con el mínimo de esfuerzo y tiempo posible (restricción).

Los casos de prueba pueden tener distintos orígenes, desde:

* Los documentos del cliente
* Información de relevamiento
* Requerimientos
* Especificaciones de programación
* El código

En las pruebas de sistema partimos desde los requerimientos, ya que si los sacara del código, puedo verificar (que funcione correctamente) pero no validar (que haga lo que tiene que hacer).

**Conclusiones sobre la generación de casos de prueba**

* Hay muchas técnicas para hacer casos de prueba, pero ninguna es completa
* Las técnicas atacan distintos problemas, pero todas tienen ventajas y desventajas
* Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una
* Es muy valiosa la experiencia de tester experimentados
* Depende del código a testear
* Sin requerimientos todo es mucho más difícil, y diría casi imposible, ya que no puedo validar, sólo verificar.
* Ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas es muy importante, ya que sin estructura (ad hoc) no es útil.

**Condiciones de prueba**

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.

**Métodos de prueba**

Existen distintos métodos, que podemos utilizar debido a que el tiempo y el presupuesto es limitado, lo que implica que debo pasar por la mayor cantidad de funcionalidades con la menor cantidad de pruebas posible

**Caja Negra**

Son métodos dinámicos donde se prueba el sistema con distintas entradas y salidas, sin analizar su funcionalidad interna. Hacemos foco en el “qué hace” y no en el “cómo lo hace”. Pueden estar basados en especificaciones o en la experiencia. Los primeros se ejecutan utilizando documentación sobre las especificaciones del producto, y dentro de él se encuentra:

* **Partición de equivalencias**: es un proceso sistemático que identifica un conjunto de clases de prueba representativas de conjuntos más grandes, y define dos pasos.
  + **1**. **Identificar las clases de equivalencias válidas y no válidas**: es un proceso sistemático donde identificamos esas clases de prueba representativas. Debemos identificar las condiciones externas involucradas en una funcionalidad, tanto las entradas como salidas posibles. Podemos hacerlo con diferentes técnicas:
    - Rango de valores contínuos: una clase válida dentro del rango y dos inválidas de cada lado fuera del rango
    - Valores discretos: una clase válida dentro del rango y dos inválidas de cada lado fuera del rango
    - Selección simple: una clase válida (en el conjunto) y una inválida (fuera del conjunto)
    - Selección múltiple: idem anterior
  + **2. Identificar los casos de prueba:** se analiza cuales son los subconjuntos de valores posibles que pueden tomar esas condiciones externas, que producen un resultado equivalente. Cada clase tendrá un número único, y debemos cubrir todas las clases tanto válidas como inválidas. Si múltiples clases inválidas se prueban con la misma prueba, algunas pueden no ejecutarse (la primera prueba enmascara a las otras).
* **Análisis de valor límite:** es una variante del anterior, donde se seleccionan los bordes de una clase como representativos, definiendo clases de equivalencia tanto de salida como de entrada.

Por otro lado, los que están basados en la experiencia son aquellos donde la experiencia de quien se ocupa de determinar las entradas es fundamental. Incluye:

* **Adivinanzas de defectos:** enfoque basado en experiencia e intuición, para identificar pruebas que probablemente tendrán defectos. Elaboramos una lista de defectos posibles o situaciones propensas a error, y desarrollamos las pruebas basadas en esa lista.
* **Testing exploratorios:** la principal característica es que el aprendizaje, diseño y ejecución de pruebas se realizan de forma simultánea (mientras el tester prueba software, aprende a manejar el sistema, y junto a esa experiencia, genera nuevas pruebas). Es fundamental el compromiso cognitivo del tester y la responsabilidad para gestionar su tiempo.

**Caja blanca**

Son métodos estáticos que se basan en analizar la estructura interna del software o componente de software. Se puede garantizar el testing coverage, es decir, diseñar casos de prueba que garanticen, por ejemplo, que todos los caminos dentro de un módulo se hayan ejecutado al menos una vez, que se ejecuten todas las decisiones lógicas en sus lados verdaderos o falsos, etc. Las razones para hacer testing de caja blanca son:

* Errores lógicos o supuestos incorrectos que son poco probables que se ejecuten
* Creencia errónea de que un camino no se ejecutará habitualmente
* Errores tipográficos
* Los errores anteriores no se pueden detectar con testing de caja negra

La **complejidad ciclomática** es una métrica que provee una medición cuantitativa de la complejidad de un programa que, en testing, define el número de caminos independientes en el conjunto básico y entrega un límite inferior para el número de casos necesarios para ejecutar todas las instrucciones al menos una vez.

Métodos:

* **Cobertura de sentencias:** recorrer todos los caminos lógicos
* **Cobertura de decisión:** recorrer cada decisión al menos una vez por ambas ramas
* **Cobertura de condición:** verificar cada condición en una decisión tenga todos los resultados posibles
* **Cobertura de decisión/condición:** verificar que cada condición en una decisión tenga todas las posibles salidas y cada punto de entrada sea activado al menos una vez
* **Cobertura múltiple:** ejecutar todas las combinaciones posibles de resultados de condición en cada decisión y todos los puntos de entrada al menos una vez.
* **Cobertura de enunciados**
* **Cobertura de loop**

Cada método tiene sus fortalezas y debilidades particulares, por lo que el mejor método es usar una variedad de técnicas en conjunto para tener un testing efectivo.

**Ciclo de test o de prueba**

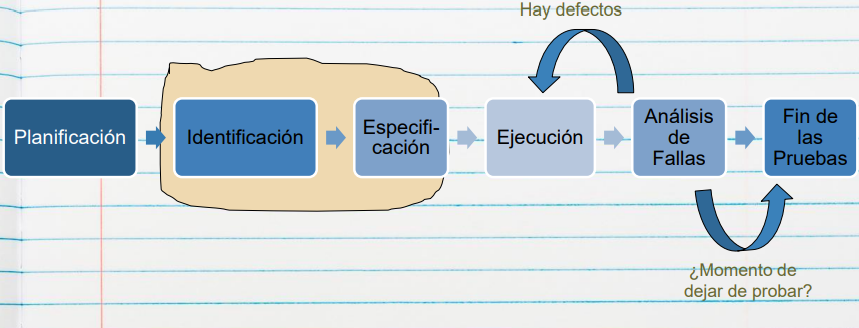
Un ciclo de test o de pruebas es la ejecución de la totalidad de los casos de pruebas establecidos aplicados a una misma versión del sistema a probar. Es decir, se realiza cuando tenemos un incremento listo para ser probado y tenemos los casos de prueba para probarlo. El primer ciclo es llamado ciclo 0, el cual siempre será manual y no lo puedo automatizar, debido a que es la primera vez que pruebo. En cambio, a partir del ciclo 1 ya se puede automatizar, y puedo definir la estrategia de como encararlo: con o sin regresión.

Al concluir un ciclo de pruebas, y reemplazarse 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. Sin embargo, esto implica un extra de costos y de tiempo, por lo que realizar ciclos de test sin regresión suele ser una opción, pero puede implicar que al intentar arreglar un error, introduzca otro en otra parte que, como ya testee anteriormente, debería funcionar correctamente, provocando un “problema escondido”, un defecto, que no podré encontrar sin regresión.

Entonces, podemos decir que es la actitud o estrategia de cómo abordar los casos de prueba a partir del ciclo 1 (ya que el ciclo 0 siempre lo hago): si lo afronto **con regresión**, voy a re-testear todos los casos de prueba que se incluyeron en el ciclo 0, y si lo afronto **sin regresión**, sólo voy a probar casos de prueba que no ejecuté anteriormente, asumiendo que las pruebas anteriores volverán a pasar (resultado correcto, no dará errores). Si yo tengo trazabilidad, es más fácil saber en qué partes del sistema puede impactar un cambio, lo que me facilita seleccionar la estrategia adecuada en cada caso. Una opción también es hacer varios casos de prueba sin regresión, y cada X ciclos hacer uno con regresión.

El costo de automatizar los casos de prueba luego se amortiza a lo largo de los ciclos de prueba, pero hay que analizar cuántos ciclos de prueba se esperan realizar, para saber si es conveniente o no automatizar. Generalmente, para software más antiguo, se tiende a realizar más ciclos de prueba, porque la estructura del sistema se va degradando con los cambios, y es más común que al reparar un error se impacte en otra parte del sistema.

**Proceso de pruebas**

Es muy importante que las pruebas tengan una estructura, sean sistemáticas, metodológicas (si realizamos pruebas “ad hoc” no tenemos proceso). Las etapas de identificación y especificación se pueden hacer juntas. Algo a destacar es que como dijimos, el testing se realiza sobre el código, lo que significa que hasta que hasta antes de la ejecución no podremos probar nada, pero no implica que no podamos hacer nada de testing, ya que podremos ir diseñando los casos de prueba con tan sólo conocer los requerimientos, puedo también realizar el plan de testing, construyendo las bases de datos para los casos de prueba, preparar el ambiente de trabajo, etc., sin necesitar ni una línea de código.

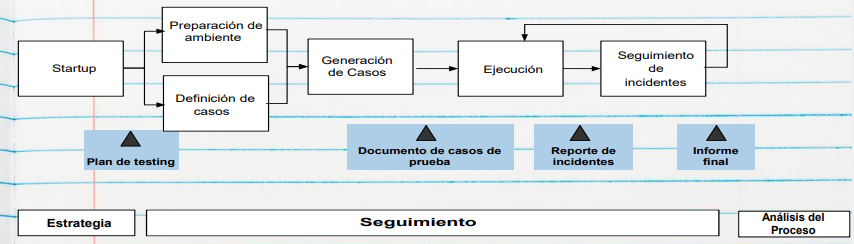
**Planificación y control:** es la actividad de verificar que se entienden las metas/objetivos del cliente y partes interesadas, el proyecto y los riesgos de las pruebas. El resultado es el Plan de Pruebas.

**Análisis y diseño de las pruebas (identificación y especificación):** se toman las entradas, se hace el análisis y se escriben los casos de pruebas. Para ello debemos revisar las base de pruebas, evaluar la testeabilidad de los requerimientos y el sistema, identificamos los datos necesarios, diseñamos y priorizamos los datos de prueba y diseñamos el entorno de prueba.

**Ejecución:** es la etapa donde se ejecutan los casos de prueba, en la prioridad que establecimos anteriormente porque los tiempos y esfuerzo son limitados. Implica crear los datos de prueba, automatizar los que sea necesario (se debe analizar su conveniencia y sus costos), se organizan los casos de pruebas en conjuntos para ejecutarlos eficientemente, se implementa y verifica el ambiente, y finalmente se ejecutan los casos de prueba, registrando los resultados y comparando los resultados reales con los esperados

**Evaluación y reporte:** en función de los resultados obtenidos, se evalúan los criterios de aceptación, se reportan los resultados a los stakeholders, se verifica que los defectos hayan sido corregidos y se evalúa qué tan eficiente fue el proceso de testing y se intenta mejorar en los próximos ciclos.

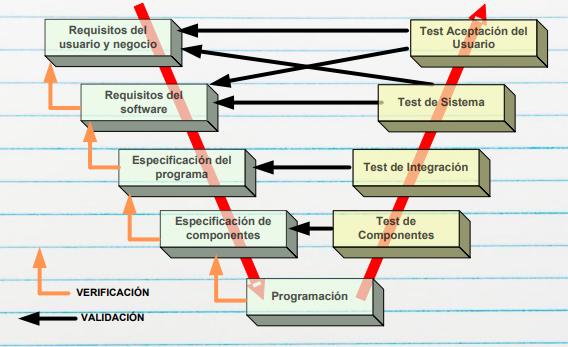
Un aspecto muy importante a definir es **cuándo dejo de probar**, ya que nunca voy a dejar de encontrar defectos. Un criterio muy útil es dejar de probar en algún momento en donde se están gastando demasiados recursos (como tiempo y esfuerzo) y se encuentran pocos defectos. En empresas maduras, se suele elegir como criterio el momento en donde no hay defectos bloqueantes o críticos/mayores conocidos, y tenemos un cierto porcentaje, lo suficientemente pequeño, de los defectos y cosméticos sin corregir.

**Artefactos de testing**

Por un lado, tenemos un conjunto de artefactos no negociables, mientras que otros son opcionales y suelen no realizarse en empresas pequeñas. Los primeros son:

* **Plan de testing**: se incluyen en él todas las decisiones de cómo vamos a realizar el testing. Siempre debo definir primero un plan antes de diseñar los casos de prueba. Contiene los riesgos y objetivos del testing, la estrategia de testing, recursos y criterios de aceptación
* **Documento de casos de prueba**: se documenta el conjunto de casos de pruebas diseñados, puede ser realizado con distintas herramientas.
* **Reporte de incidentes**: es el conjunto de defectos encontrados, y puede ser realizado con distintas herramientas como Gira o incluso con Excel.

El artefacto de testing opcional es el **informe final**, donde se incluyen los resultados de las métricas, información contextual, cantidad de ciclos de prueba realizados, condiciones de ejecución de pruebas, los tipos de prueba, la clasificación y priorización de defectos, la estrategia utilizada (con o sin regresión), entre otros. A pesar de que sea un artefacto opcional, si el área de testing tiene el peso necesario como para poder frenar la salida de un release debido a la existencia de muchos defectos en el incremento, es muy necesario tenerlo. Esto sucede mayormente en empresas grandes, en situaciones en donde está en juego mucho dinero o situaciones críticas (por ejemplo, en bancos).

**Verificación y validación:** en testing realizamos las dos cosas, es decir, **validamos**, determinando si estamos construyendo el sistema que el cliente quiere, y **verificamos,** porque determinamos si estamos construyendo el sistema correctamente.

El modelo en V nos muestra que, en el testing, se realizan las actividades en sentido inverso al desarrollo: de lo más particular a lo más general, ya que partimos por los componentes en pruebas unitarias, luego la integración, testeamos el sistema y luego las pruebas de aceptación (relacionado a los requerimientos de usuario). También podemos observar que sólo podemos hacer validación cuando tenemos código, porque implica contrastar este último con los requerimientos.

Comentarios de clase: Una forma de especificar requerimientos de usuario es con user stories, mientras que los requerimientos de software se especifican con casos de uso. Con especificación de programa nos referimos a la arquitectura, y con especificación de componentes nos referimos a un detalle más fino.

**El testing en el ciclo de vida del software**

Siempre es conveniente involucrar las actividades de testing de manera temprana, para dar visibilidad de manera temprana al equipo de cómo se va a probar el producto y disminuir los costos de correcciones de defectos. Es por eso que si tuviésemos un ciclo de vida en cascada, es difícil avanzar tempranamente con el testing, ya que necesitaríamos tener una ERS completa antes de continuar, porque necesito mínimamente los requerimientos para poder construir los casos de prueba, lo que implica que sólo podré hacer testing con el 100% del producto terminado. En cambio, con un ciclo de vida iterativo, puedo testear los incrementos resultantes de cada iteración, diseñando los casos de prueba sólo para los casos de uso implementados; aunque la dificultad es que, como siempre se realizan incrementos del producto anterior, nunca voy a estar seguro de que lo que antes probé seguirá funcionando correctamente, por lo que se debe invertir un mayor esfuerzo debido a esta integración (hay que utilizar regresión).

**Romper mitos**

* El testing es una etapa que comienza al terminar de codificar: como vimos, se puede comenzar mucho antes a diseñar los casos de prueba a partir de requerimientos
* El testing es probar que el software funciona: en realidad es, a partir de asumir que el software tiene defectos, intentar encontrarlos.
* Testing = calidad de producto: no suma calidad, solo pone en evidencia la falta de ella
* Testing = calidad del proceso
* El tester es el enemigo del programador: el encontrar defecto permite proteger al programador para no dejar en evidencia los fallos ante el cliente.

**¿Cuánto testing es suficiente?**

El testing exhaustivo es imposible, pero ¿hasta donde lo hacemos?. Una gran ayuda para determinarlo son los **criterios de aceptación,** porque son concretos, y a partir de ellos se negocia con el cliente (particularmente con el Product Owner) hasta donde debemos probar. Muchas veces no son suficientes pero es una buena base para saber cuando una user story está lo suficientemente testeada. Puede tener un mismo caso de prueba que tenga en cuenta varios criterios de aceptación.

Un error sería determinar el límite del testing mediante fallas predichas, por ejemplo, “frenar cuando encuentre 15 fallas”, debido a que es posible que sólo encuentre 10 y nunca paro de testear siendo un desperdicio de tiempo importante, o caso contrario, que sean demasiado pocos y me queden demasiados defectos sin corregir.

**Principios del testing**

* El testing muestra presencia de defecto (no la ausencia de ellos)
* El testing exhaustivo es imposible
* Testing temprano
* Agrupamiento de defectos (el 80% de defectos se agrupa en el 20% de la funcionalidad)
* Paradoja del pesticida (si utilizo siempre el mismo pesticida, los insectos generan una cierta resistencia, lo mismo sucede con el testeo: si no cambio a los testers a veces, será más difícil encontrar defectos porque no saldrán del molde, no podrán ver las cosas de diferente perspectiva)
* El testing es dependiente del contexto (es posible que un software funcione en una PC pero no en otra, por ejemplo)
* Falacia de la ausencia de errores
* Un programador debería evitar probar su propio código
* Una unidad de programación no debería probar sus propios desarrollos
* 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.

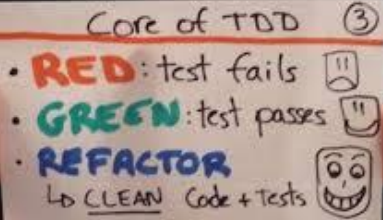
**La psicología del testing:** muchas veces, la búsqueda de fallas puede ser vista como una crítica al producto y/o a su autor porque no comprenden que el trabajo del tester es encontrar defectos. Además, la construcción del software requiere otra mentalidad a la de testear software, ya que un desarrollador generalmente es poco detallista y muy optimista (“¿Por que el usuario haría eso?”), mientras que un tester debe ser más detallista y pesimista (intenta probar el mal funcionamiento)

**Tipos de prueba**

**Smoke test:** es un tipo de prueba que me permite comprobar si existen fallas groseras en la versión del producto a testear, es decir, es la primer corrida de los test de sistema que provee cierto aseguramiento de que el software que está siendo testeado no provoca una falla catastrófica. Es una prueba de grano grueso o corrida rápida para saber si está en condiciones de entrar a testing, antes de la ejecución de un ciclo de prueba.

**Testing funcional:** las pruebas se basan en funciones y caracteristicas (descripta en los documentos o entendidas por los testers) y su interoperabilidad con sistemas específicos, y puede estar basado en requerimientos (pruebo cada CU o US por separado, en particular) o basado en los procesos de negocio (pruebo un conjunto de US de un proceso completo de negocio).

**Testing no funcional:** es una prueba de cómo funciona el sistema, es decir, de requerimientos no funcionales. Suele ser difícil de probar porque dependen mucho del contexto, por lo que puedo intentar reproducir el entorno/ambiente de producción, pero no tengo la certeza de que funcione en el ambiente de producción real. Muchas veces no se hace. Algunas de las pruebas que incluye son:

* Performance testing: que el tiempo de respuesta no supere un cierto compromiso asumido
* Pruebas de carga: no solo mido tiempos de respuesta sino otros parámetros, a carga normal
* Pruebas de stress: fuerzo al sistema más allá de lo esperado, para saber qué sucede y en cuanto tiempo puedo recuperar el sistema
* Pruebas de usabilidad: de experiencia de usuario
* Pruebas de mantenimiento: si el producto puede ser evolucionado (manuales, documentación, facilidad de encontrar la ubicación de defectos, etc.)
* Pruebas de fiabilidad: resultados, funcionamiento, disponibilidad
* PRuebas de portabilidad: si funciona en otros entornos o navegadores

**TDD:** el Desarrollo Guiado por Pruebas de Software es una técnica avanzada que involucra otras dos prácticas: escribir las pruebas primero (Test first Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias. La idea es que el diseñar test es un mecanismo muy efectivo para prevenir errores, porque permite hacer un “cambio de chip” o cambio de perspectiva, ya que el proceso mental que hay que desarrollar para crear test útiles me permite descubrir y eliminar problemas en todas las etapas del desarrollo.