**Principios de diseño de Smalltalk**

**Buen diseño** un sistema debería ser construido con un mínimo conjunto de partes no modificables tan generales como sea posible.

Un sistema debe ser completamente entendible para un individuo.

Cualquier parte que no pueda ser cambiada o no es lo suficiente general, es probablemente un origen de impedimentos.

**Lenguaje** Su propósito es proveer un esquema para la comunicación.

El diseño de un lenguaje para usar computadoras debe tratar con modelos internos, medios externos y la interacción entre ellos tanto en el humano como en la computadora.

**Objetos** entidad única que la solución. Mensaje es la forma de comunicación entre ellos.

La administración del almacenamiento debe ser automatizada. La abstracción debe ser aplicable en todas las áreas del sistema.

**Modularidad** ningún componente en un sistema complejo debería depender de los detalles internos de ningún otro (encapsulamiento)

Se debe clasificar estos objetos mediante clases.

**Polimorfismo** un programa solo debería especificar el comportamiento esperado de los objetos, no su representación.

**Factorización** cada componente independiente de un sistema solo debería aparecer en un solo lugar (herencia)

**Maquina virtual** establece un marco para la aplicación de tecnología.

**Selección natural** los lenguajes y el sistema de buen diseño persistirán solo para ser reemplazados por otros mejores.

**PAUTAS PARA TEST UNITARIOS.**

1. Mantenerlos pequeños y rápidos. Esto reduce el tiempo de respuesta de desarrollo.
2. Deben ser automatizadas y no interactiva.
3. Simples para ejecutar.
4. Hay que medir los test para que sea posible leer la ejecución de cobertura exacta e investigar que parte del código se ejecuta y cual no.
5. Reparar las pruebas que fallan de inmediato.
6. Mantener las pruebas unitarias. Probar por clases de forma aislada.
7. Comenzar simple. Una clase de prueba simple establecerá el marco de la clase objetivo.
8. Test independientes, asegura la robustez y la facilidad de mantenimiento de las pruebas. No deben depender del orden ni de otras.
9. Mantener las pruebas cerca de la clase a prueba.
10. Nombrar los test de forma apropiada.
11. Probar API públicas.
12. Pensar como una caja negra, hay que pensar como consumidor e intentar hacer que el código se rompa (pruebas unitarias)
13. Piensa como una caja banca, hay que probar la lógica, ver el código en sí (debbugger)
14. Prueba las clases triviales. Lo trivial es difícil de definir para otra persona que intenta romper nuestro código, estas pueden contener errores también.
15. Concéntrate en la ejecución de cobertura primero.
16. Cubrir casos límites, son los principales errores en los programas.
17. Proveer un generador aleatorio, permite probar diferentes valores de entradas de forma barata.
18. Probar una característica a la vez, esto hará la mantención mucho más fácil. Colocar el nombre de esta en el método de la prueba.
19. Usar afirmaciones explícitas.
20. Pruebe test negativos.
21. Diseña el código con las pruebas en mente.
22. No conectarse a recursos externos indefinidos.
23. Conozca el costo de las pruebas
24. Prioriza las pruebas.
25. Prepara la prueba para los errores, que al encontrar uno no corte la ejecución de las siguientes.
26. Escribe pruebas para reproducir errores.
27. Conozca las limitaciones, las pruebas unitarias jamás probarán con exactitud el código

**8 PRINCIPIOS PARA MEJORES**

**PRUEBAS UNITARIAS.**

Las pruebas unitarias son cortas, rápidas y automáticas que aseguran que una parte específica del programa funciona.

1. Hay que entender lo que se está probando. Testear solo una cosa crea una prueba más legible ya que cuando falla es más fácil encontrar el error y arreglarlo.
2. Deben ser autosuficientes. Evitar que le afecte el orden de ejecución y que no dependan de otras.
3. Deben ser deterministas. O fallan siempre o pasan siempre, nunca puede ser a medias. Otra forma de evitarlo es no crear pruebas aleatorias ya que crea incertidumbre, ¿Cómo sabemos cual de todas ellas falló?
4. Buenas convenciones de nombre.
5. Do repeat yourself. No importa si hay código repetido, es importante que las pruebas sean legibles.
6. Testea los resultados, no la implementación. Si se desea realizar cambios en el código las pruebas tendrían que seguir funcionando igual.
7. Evitar la sobre especificación.
8. Use un marco de aislamiento.

**THE ART OF ENBUGGING**

¿Cómo se pueden prevenir bugs? Una buena forma es mantener la “separación de preocupaciones” apropiadas. Esto quiere decir diseñar el código para que las clases y los módulos estén limpios, bien definidos con responsabilidades aisladas y semántica entendible.

La meta fundamental es escribir un código que no revele mucho de él y que no hable con otro más que lo necesario, a esta definición el autor refiere cuando expresa sobre un “código tímido”.

Formas para crear esta clase de código:

**Tell, don’t ask.** Se debe decir a los objetos que hagan algo, no hacer cosas por ellos. No queremos preguntarle explícitamente a un objeto sobre su estado. El objeto que llama no debe modificar el estado del otro, este accionar viola el concepto de encapsulamiento y termina creando bugs. La separación entre un comando y la consulta hace a un código “tímido”.

**La buena idea de Demetrer.** Mientras se llame a más objetos, más se incrementa la posibilidad de romper un código cuando se cambia a un objeto. Esto sugiere que un objeto solo debe llamarse a si mismo, a cualquier parámetro que se le haya pasado en el método, a cualquier objeto que haya creado y a cualquier componente del objeto contenido directamente.

**GETTERERADICATOR**

El punto del encapsulamiento no es ocultar la información sino ocultar las decisiones del diseño que en el futuro pueden cambiar. Lo mejor es que, al pensar ello, te preguntes ¿qué piezas de variabilidad escondes y por qué? Más qué ¿Estoy exponiendo datos?

Evitar los get/set ya que violan el principio Tell, don’t ask. Dejarlos por completo es más contundente porque puede existir casos en donde nos obligue su aplicación al no poder resolver el problema de otra forma.

Si se busca una regla de oro, esta puede ser “Siempre tener cuidado con los casos en el que algún código invoca más de un método a un mismo objeto”. ¿No se lo puede reemplazar todo en un solo comando? Hay casos en que no se puede, pero igual es valioso hacerse la pregunta antes.

Otra señal de advertencia es la clase de datos, una clase que solamente tiene campos y accesores. Es casi siempre una señal de problema porque carece de comportamiento.

No hay una guía para la asignación de los comportamientos de cada objeto, pero una buena práctica es poner el comportamiento en la misma clase que la información, pero no es la única.

Una buena regla de oro es: Las cosas que cambian juntas, deberían ir juntas.

**PRUEBAS DE SOFTWARE.**

**Verificación:** construir que hayamos construido el código tal como lo pensamos. **Validación:** que el mismo sea lo que el cliente esperaba.

Las **pruebas unitarias** verifican una pequeña porción de código. En el paradigma de objetos verifican alguna responsabilidad única del objeto. Posee la ventaja de permite aislarse del conjunto del sistema y verificar que una pequeña porción de este funcione de la manera adecuada

Las **pruebas de integración** verifican que varias porciones del código, trabajando en conjunto, hacen lo que pretendíamos. Estas no siempre prueban el sistema como un todo, a veces se prueban partes de este que se desea aislar

**Pruebas de verificación**

**Caja negra** lo miramos sin ejecutar el código que estamos probando, en caso contrario se lo analiza como una **caja blanca**. Generalmente se prefieren pruebas de caja negra.

**Alcance de las pruebas de verificación.**

Las **pruebas de aceptación de usuarios (UAT)** se deben ejecutar en un entorno lo más parecido posible al que va a utilizar el usuario

**Pruebas Alfa** usuarios prueban el programa, pero en un ambiente controlado por el equipo de desarrollo. **Pruebas Beta** el cliente lo prueba en su entorno.

En estas pruebas se suponen que ejecutamos el sistema completo. Si solo queremos probar el comportamiento, en el sentido si la lógica de la aplicación es correcta, pero sin la interfaz del usuario, estas pruebas se denominan **pruebas de comportamiento.**

**Pruebas en producción.**

Poner un sistema en producción y ver que errores surgen o los problemas que los usuarios encuentren. Esto resulta costoso ya que se los solucionaría de forma tardía. En cambio, si estos errores no traen riesgos, podría ser una opción.

**Pruebas de regresión.**

Cada vez que incorporamos una nueva característica al programa podemos provocar que dejen de funcionar otras que ya habían sido probadas y entregadas (regresión). Esto no es más que ejecutar todas las pruebas del sistema a intervalos regulares.

**Ventajas de la automatización:**

1. Nos independizan del factor humano.
2. Más fácil repetir las mismas pruebas.
3. Las pruebas de código sirven como herramientas de comunicación, minimalista ambigüedades.

**WHAT’S A MODEL FOR?**

El desarrollo de UML es un importante avance. Sin embargo, a los clientes no les interesa: ellos solo quieren ver el software. Si el modelo mejora la calidad o reduce el costo del software, entonces tiene valor. Modelar lleva tiempo. Existen herramientas que transforman texto en diagramas. Sin embargo, hay que ser cuidadoso con el nivel de detalle que se incluyen en los diagramas. Conviene usar un “modelo esquelético” que muestre los huesos del sistema. Este modelo es más fácil de mantener, porque los detalles importantes no cambian seguido. Pero este tipo de modelo hay que hacerlo uno mismo, pues las computadoras no saben distinguir los detalles que son importantes de los que no lo son.

Funciones de un modelo:

1) Dar mayor entendimiento del problema

2) Resaltar detalles importantes: modelo esquelético