Tema 1: Introducción al Testing de Programas

Miguel Gómez-Zamalloa Gil

Dep. de Sistemas Informáticos y Computación. UCM

Especificación, Validación y Testing
Optativa de los Grados de la Facultad de Informática,
Curso 2015-2016

Índice

Introducción al Testing de Programas

- Definiciones, importancia, evolución e historia
- Un primer ejemplo
- Psicología del testing
- Principios del testing de programas
- Clases de testing de programas
- Testing dinámico vs. testing estático
- Black-box testing vs. white-box testing
- Niveles de testing

Definiciones

Definición "The Art of Software Testing" (ArtOfST)

Serie de procesos orientados a asegurar que el software hace lo que debe hacer y no hace lo que no debe

Definión Wikipedia

Investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto a la parte interesada

Definiciones

Definición IEEE

Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item

Definión Dave Gelperin y William C. Hetzel

Process of validating and verifying that a software program/application/product: 1. meets the requirements that guided its design and development 2. works as expected 3. can be implemented with the same characteristics 4. satisfies the needs of stakeholders

Importancia del testing

En ArtOfST...

Cuando se publicó el libro en 1979, una regla bien establecida decía que más del 50 % de los costes de desarrollo de software se empleaban en el testing de programas. Hoy, en el 2004, esto se sigue cumpliendo.

- Ha habido avances extraordinarios en el desarrollo de lenguajes, técnicas y herramientas, pero el testing sigue siendo una de las areas más difusas.
- Se considera una de las "artes oscuras" en el desarrollo de software.

Evolución del testing

En ArtOfST...

Desde que se publicó este libro en 1979, el testing se ha convertido en más fácil y al mismo tiempo mas difícil que nunca.

- Más difícil debido a:
  - Gran variedad de sistemas operativos, hardwares, lenguajes y paradigmas de programación
  - Complejidad del software (paralelismo/concurrencia, grid/cloud computing)
- Más fácil gracias a:
  - Rutinas y librerías de funciones bien documentadas y testeadas
  - Herramientas que ayudan al testing como depuradores gráficos, frameworks de unit-testing (jUnit), verificadores y analizadores, etc.
  - Metodologías de desarrollo de software (patrones, desarrollo de software ágil - "agile software development")

Historia del testing

- http://www.testingreferences.com/testinghistory.php
- http://www.softwaretestpro.com/Item/4537/History-of-Ideas-in-Software-Testing/

Un primer ejemplo

Función TipoTriangulo tipoDeTriangulo(int,int,int)

Programa que recibe tres números enteros que representan las longitudes de los lados de un triángulo, y devuelve un valor en el conjunto {invalido, equilatero, isosceles, escaleno} indicando el tipo de triángulo (y si es o no válido).

Un primer ejemplo

Función TipoTriangulo tipoDeTriangulo(int,int,int)

Programa que recibe tres números enteros que representan las longitudes de los lados de un triángulo, y devuelve un valor en el conjunto {invalido, equilatero, isosceles, escaleno} indicando el tipo de triángulo (y si es o no válido).

Un buen conjunto de tests debería considerar los siguientes casos:

1. Caso que represente un triángulo escaleno válido. Ojo, que $\langle 1, 2, 3 \rangle$ o $\langle 2, 5, 10 \rangle$ no debería ser considerados triángulos escalenos válidos
2. . . . triángulo isósceles válido ($\langle 2, 2, 4 \rangle$ no es válido!)
3. . . . triángulo equilatero válido
4. Permutaciones de lados en triángulos isósceles

Un primer ejemplo

5. Un lado es 0. Permutaciones
6. Lado negativo. Permutaciones
7. Tres enteros mayores que cero tal que la suma de dos de ellos es menor que el mayor. Permutaciones.
8. Tres enteros mayores que cero tal que la suma de dos de ellos es igual al mayor. Permutaciones.
9. Todos los lados son 0

- Si la entrada se leyese por teclado (o fichero) habría que comprobar que son enteros válidos, y que se reciben exactamente tres

Psicología del testing

- Una de las causas principales de un mal testing es basarse en las habituales definiciones de testing. E.g:

"El testing es el proceso de demostrar que no hay errores"

o bien,

"El próposito del testing es mostrar que un programa se comporta como se espera"

o bien,

"El testing es el proceso por el cual se establece confianza de que un programa hace lo que debe hacer"

Psicología del testing

- Basarse en estas definiciones es un error
- No trates de mostrar que un programa funciona bien. **Asume que tiene errores** (válido para casi todos los programas) y **trata de encontrar el mayor número de errores**
- Una definición más apropiada de testing sería:

> **El testing es el proceso de ejecutar un programa con el objetivo de encontrar errores en él**

- El testing puede verse como un proceso destructivo
- Esto tiene implicaciones sobre quién y cómo se debe realizar el testing
- Test *exitoso* vs. test *fallido* (*successful* vs. *unsuccessful test*)

Psicología del testing

- Otro problema al definir el objetivo equivocado es que éste se convierte en virtualmente inalcanzable
  - Los estudios demuestran que el ser humano no es capaz de rendir bien cuando se enfrenta a objetivos inalcanzables
  - Encontrar fallos en el software si parece algo factible
- Ahora bien, ¿es posible encontrar todos los errores de un programa?
- La respuesta en general es negativa, bien imposible o bien inviable
- Hay que establecer por tanto algunos criterios y estrategias

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

1. Una parte necesaria de un test es la definición precisa del resultado esperado
   - "El ojo ve lo que quiere ver"

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

1. Una parte necesaria de un test es la definición precisa del resultado esperado
   - "El ojo ve lo que quiere ver"
2. Un programador no debe testear sus propios programa
   - Difícil ver ciertos errores. Difícil pasar a ser destructivo

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

1. Una parte necesaria de un test es la definición precisa del resultado esperado
   - "El ojo ve lo que quiere ver"
2. Un programador no debe testear sus propios programa
   - Difícil ver ciertos errores. Difícil pasar a ser destructivo
3. Una compañía no debe testear sus propios programas
   - La compañía está fuertemente guiada por objetivos, fechas y costes

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

1. Una parte necesaria de un test es la definición precisa del resultado esperado
   - "El ojo ve lo que quiere ver"
2. Un programador no debe testear sus propios programa
   - Difícil ver ciertos errores. Difícil pasar a ser destructivo
3. Una compañía no debe testear sus propios programas
   - La compañía está fuertemente guiada por objetivos, fechas y costes
4. Los resultados de cada test deben ser inspeccionados a fondo
   - Errores encontrados en tests son habitualmente pasados por alto en tests previos

Principios del testing de programas

Se enumeran a continuación 10 principios fundamentales. Aunque algunos resultan obvios es habitual pasarlos por alto.

1. Una parte necesaria de un test es la definición precisa del resultado esperado
   - "El ojo ve lo que quiere ver"
2. Un programador no debe testear sus propios programa
   - Difícil ver ciertos errores. Difícil pasar a ser destructivo
3. Una compañía no debe testear sus propios programas
   - La compañía está fuertemente guiada por objetivos, fechas y costes
4. Los resultados de cada test deben ser inspeccionados a fondo
   - Errores encontrados en tests son habitualmente pasados por alto en tests previos
5. Se deben escribir también tests para condiciones no válidas de entrada

Principios del testing de programas

6. A parte de buscar casos en los que el programa no hace lo que debe hacer, hay que buscar casos en los que **hace lo que no debe hacer**.
   - Ojo con los *efectos colaterales*. Ej.: Un programa que realiza correctamente transferencias bancarias será erroneo si produce cambios sobre usuarios no involucrados en la transferencia.

Principios del testing de programas

6. A parte de buscar casos en los que el programa no hace lo que debe hacer, hay que buscar casos en los que **hace lo que no debe hacer**.
   - Ojo con los *efectos colaterales*. Ej.: Un programa que realiza correctamente transferencias bancarias será erroneo si produce cambios sobre usuarios no involucrados en la transferencia.
7. Evitar realizar tests *volátiles* salvo que no haya otra posibilidad
   - Escribir tests es una inversión (por ej. para *regression* testing)

Principios del testing de programas

6. A parte de buscar casos en los que el programa no hace lo que debe hacer, hay que buscar casos en los que **hace lo que no debe hacer**.
   - Ojo con los *efectos colaterales*. Ej.: Un programa que realiza correctamente transferencias bancarias será erroneo si produce cambios sobre usuarios no involucrados en la transferencia.
7. Evitar realizar tests *volátiles* salvo que no haya otra posibilidad
   - Escribir tests es una inversión (por ej. para *regression* testing)
8. Nunca se debe asumir que no se encontrarán errores

Principios del testing de programas

6. A parte de buscar casos en los que el programa no hace lo que debe hacer, hay que buscar casos en los que **hace lo que no debe hacer**.
   - Ojo con los *efectos colaterales*. Ej.: Un programa que realiza correctamente transferencias bancarias será erroneo si produce cambios sobre usuarios no involucrados en la transferencia.
7. Evitar realizar tests *volátiles* salvo que no haya otra posibilidad
   - Escribir tests es una inversión (por ej. para *regression* testing)
8. Nunca se debe asumir que no se encontrarán errores
9. La probabilidad de existencia de más errores en una parte del programa es proporcional al número de errores ya encontrado en ella
   - Los errores tienden a venir agrupados. Además es habitual arreglar una cosa y romper otra.

Principios del testing de programas

6. A parte de buscar casos en los que el programa no hace lo que debe hacer, hay que buscar casos en los que **hace lo que no debe hacer**.
   - Ojo con los *efectos colaterales*. Ej.: Un programa que realiza correctamente transferencias bancarias será erroneo si produce cambios sobre usuarios no involucrados en la transferencia.
7. Evitar realizar tests *volátiles* salvo que no haya otra posibilidad
   - Escribir tests es una inversión (por ej. para *regression* testing)
8. Nunca se debe asumir que no se encontrarán errores
9. La probabilidad de existencia de más errores en una parte del programa es proporcional al número de errores ya encontrado en ella
   - Los errores tienden a venir agrupados. Además es habitual arreglar una cosa y romper otra.
10. El testing es una tarea extremadamente creativa y desafiante
    - Probablemente la creatividad requerida para testear adecuadamente un programa es mayor a la requerida para implementarlo

Clases de testing de programas

- Hay decenas de términos que hacen referencia a distintas clases de testing de programas
  - Static testing, dynamic testing, black-box testing, white-box testing, unit testing, integration testing, system testing, acceptance testing, regression testing, installation testing, compatibility testing, security testing, performance testing, recovery testing, extreme testing, agile testing ...
- Trataremos de dar una panorámica general y una categorización de todas estas clases de testing

Clases de testing de programas

- En función de si se ejecuta o no el programa:
  1. Testing estático ⇒ El programa no se ejecuta (ej. inspecciones, revisiones, verificación, análisis estático, . . .)
  2. Testing dinámico ⇒ El programa se ejecuta (ej. unit testing clásico)

Clases de testing de programas

- En función de si se ejecuta o no el programa:
  1. Testing estático ⇒ El programa no se ejecuta (ej. inspecciones, revisiones, verificación, análisis estático, . . .)
  2. Testing dinámico ⇒ El programa se ejecuta (ej. unit testing clásico)
- En función de qué información se usa:
  1. Testing de *caja-negra* (black-box testing) ⇒ Se usa la *especificación*
  2. Testing de *caja-blanca* (white-box testing) ⇒ Se usa el *código fuente*

Clases de testing de programas

- En función de si se ejecuta o no el programa:
  1. Testing estático ⇒ El programa no se ejecuta (ej. inspecciones, revisiones, verificación, análisis estático, . . .)
  2. Testing dinámico ⇒ El programa se ejecuta (ej. unit testing clásico)
- En función de qué información se usa:
  1. Testing de *caja-negra* (black-box testing) ⇒ Se usa la *especificación*
  2. Testing de *caja-blanca* (white-box testing) ⇒ Se usa el *código fuente*
- En función del nivel en el proceso de desarrollo de software:
  1. Testing de unidad (unit testing)
  2. Testing de integración (integration testing)
  3. Testing de sistema (system testing)
  4. Testing de aceptación (acceptance testing)

Testing estático vs. dinámico

Testing dinámico

- El programa (o una parte de él) se ejecuta con una serie de casos de prueba
- Esto se hace a todos los niveles, normalmente empezando por el nivel de unidad y llegando hasta el de sistema y aceptación
- Se utiliza tanto white-box como black-box testing, aunque dependiendo del nivel suele predominar más uno u otro
- La dificultad está en diseñar casos de prueba de calidad
- Es habitual utilizar como ayuda técnicas de testing estático

Testing estático

1. Testing estático *manual* (*human testing*):
   (a) Inspecciones de código
   (b) *Walkthroughs* o *simulaciones a mano*

Testing estático

1. Testing estático *manual* (*human testing*):
   (a) Inspecciones de código
   (b) *Walkthroughs* o *simulaciones a mano*

2. Testing estático automático o semi-automático:
   (a) Análisis estático
   - Chequeo sintáctico y de tipos
   - Análisis de flujo de datos
   - Ejecución simbólica
   - Análisis de variables vivas
   - Análisis de coste, ...

Testing estático

1. Testing estático *manual* (*human testing*):
   (a) Inspecciones de código
   (b) *Walkthroughs* o *simulaciones a mano*

2. Testing estático automático o semi-automático:
   (a) Análisis estático
      - Chequeo sintáctico y de tipos
      - Análisis de flujo de datos
      - Ejecución simbólica
      - Análisis de variables vivas
      - Análisis de coste, ...
   (b) Verificación
      - Chequeo de modelos
      - Demostración automática de teoremas, ...

Inspecciones de código

- Equipo de inspección.
  - Normalmente de 4 personas: Un moderador, el programador del código, el diseñador y un experto en testing
  - El moderador lidera el proceso y anota los errores encontrados

Inspecciones de código

- Equipo de inspección.
  - Normalmente de 4 personas: Un moderador, el programador del código, el diseñador y un experto en testing
  - El moderador lidera el proceso y anota los errores encontrados
- Se realizan tres actividades:
  1. El equipo recibe el código con antelación para familiarizarse con él.
  2. El programador narra el programa instrucción a instrucción. Se hacen preguntas y se comentan.
  3. Se analiza el programa respecto a una lista de errores frecuentes
- Debe establecerse una actitud adecuada $\Rightarrow$ El programador se suele sentir atacado y adopta una postura a la defensiva

Inspecciones de código. Lista de errores frecuentes

1. Errores de referencias de datos:
   - Inicialización adecuada de variables?
   - Índices de arrays (numéricos y en el rango adecuado)?
   - Inicialización y valor de punteros?
   - *Aliasing* de referencias?, ...

Inspecciones de código. Lista de errores frecuentes

1. Errores de referencias de datos:
   - Inicialización adecuada de variables?
   - Índices de arrays (numéricos y en el rango adecuado)?
   - Inicialización y valor de punteros?
   - *Aliasing* de referencias?, ...
2. Errores de declaraciones de datos:
   - Declaraciones de variables correctas?
   - Nombres adecuados?

Inspecciones de código. Lista de errores frecuentes

1. Errores de referencias de datos:
   - Inicialización adecuada de variables?
   - Índices de arrays (numéricos y en el rango adecuado)?
   - Inicialización y valor de punteros?
   - *Aliasing* de referencias?, ...
2. Errores de declaraciones de datos:
   - Declaraciones de variables correctas?
   - Nombres adecuados?
3. Errores de cómputo:
   - Cómputos entre variables de tipos distintos? Castings automáticos?
   - Resultados fuera de rango?
   - Divisiones y módulos por cero?
   - Suposiciones sobre precedencia y asociatividad de operadores?

Inspecciones de código. Lista de errores frecuentes

1. Errores de referencias de datos:
   - Inicialización adecuada de variables?
   - Índices de arrays (numéricos y en el rango adecuado)?
   - Inicialización y valor de punteros?
   - *Aliasing* de referencias?, ...
2. Errores de declaraciones de datos:
   - Declaraciones de variables correctas?
   - Nombres adecuados?
3. Errores de cómputo:
   - Cómputos entre variables de tipos distintos? Castings automáticos?
   - Resultados fuera de rango?
   - Divisiones y módulos por cero?
   - Suposiciones sobre precedencia y asociatividad de operadores?
4. Errores de comparación:
   - Comparación de diferentes tipos? Castings automáticos?
   - Expresiones booleanas correctas? $<$ vs. $\leq$, *or* vs. *and*, etc.
   - Suposiciones de compilación *shortcut*?

Inspecciones de código. Lista de errores frecuentes

5. Errores de flujo de control:
   - Terminación de bucles?
   - Convergencia de funciones recursivas?
   - Aperturas/cierres de bloques?
   - Número correcto de iteraciones? Última iteración correcta?

Inspecciones de código. Lista de errores frecuentes

5. Errores de flujo de control:
   - Terminación de bucles?
   - Convergencia de funciones recursivas?
   - Aperturas/cierres de bloques?
   - Número correcto de iteraciones? Última iteración correcta?
6. Errores de interfaz:
   - Número, tipo y orden de argumentos?
   - Parámetros de entrada vs. parámetros de entrada/salida?

Inspecciones de código. Lista de errores frecuentes

5. Errores de flujo de control:
   - Terminación de bucles?
   - Convergencia de funciones recursivas?
   - Aperturas/cierres de bloques?
   - Número correcto de iteraciones? Última iteración correcta?
6. Errores de interfaz:
   - Número, tipo y orden de argumentos?
   - Parámetros de entrada vs. parámetros de entrada/salida?
7. Errores de entrada/salida:
   - Apertura y cierre de ficheros?
   - Condiciones de error de E/S y de final de fichero tratadas correctamente?
   - Formato de datos? Juego de caracteres?

Walkthroughs (Simulaciones manuales de código)

- Equipo de trabajo.
  - Normalmente de 3 a 5 personas: moderador, programador, secretario, experto en testing (*tester*), experto en el lenguaje de programación, etc.

Walkthroughs (Simulaciones manuales de código)

- Equipo de trabajo.
  - Normalmente de 3 a 5 personas: moderador, programador, secretario, experto en testing (*tester*), experto en el lenguaje de programación, etc.
- Se realizan las siguientes actividades:
  1. El equipo recibe el código con antelación para familiarizarse con él.
  2. El *tester* prepara una serie de casos de prueba en papel
  3. En la reunión el programa se simula manualmente para cada caso de prueba, anotando estados, trazas, errores, etc.
- De nuevo, la actitud es fundamental $\Rightarrow$ Los comentarios deben hacerse sobre el programa, no sobre el programador.

Clases de testing de programas

- En función de si se ejecuta o no el programa:
  1. Testing estático ⇒ El programa no se ejecuta (ej. inspecciones, revisiones, verificación, análisis estático, . . .)
  2. Testing dinámico ⇒ El programa se ejecuta (ej. unit testing clásico)
- En función de qué información se usa:
  1. Testing de *caja-negra* (black-box testing) ⇒ Se usa la *especificación*
  2. Testing de *caja-blanca* (white-box testing) ⇒ Se usa el *código fuente*
- En función del nivel en el proceso de desarrollo de software:
  1. Testing de unidad (unit testing)
  2. Testing de integración (integration testing)
  3. Testing de sistema (system testing)
  4. Testing de aceptación (acceptance testing)

Testing de caja-negra

- El programa se trata como una caja-negra
- Hay que concentrase en el comportamiento entrada/salida del programa, chequeando cada caso respecto a la especificación
  - Nos olvidamos por completo de la estructura y comportamiento interno del programa
- Supongamos que queremos encontrar todos los posibles errores:
  - Habría que probar con todas las posibles entradas, válidas y no válidas
  - La cantidad de casos válidos puede ser finita (ej. rango tipo int), aunque astronómica
  - La cantidad de casos no válidos suele ser infinita

Testing de caja-negra

- El programa se trata como una caja-negra
- Hay que concentrase en el comportamiento entrada/salida del programa, chequeando cada caso respecto a la especificación
  - Nos olvidamos por completo de la estructura y comportamiento interno del programa
- Supongamos que queremos encontrar todos los posibles errores:
  - Habría que probar con todas las posibles entradas, válidas y no válidas
  - La cantidad de casos válidos puede ser finita (ej. rango tipo int), aunque astronómica
  - La cantidad de casos no válidos suele ser infinita

Ejemplo: Función tipoTriangulo

- Entradas válidas: input $\in int \times int \times int$
- Nº de casos: $2^{32} * 2^{32} * 2^{32} \simeq 10^{28}$
- Si la entrada viniese de teclado o fichero sería mucho peor

Testing de caja-negra

Ejemplo: Mezcla de listas enlazadas ordenadas

- Entradas válidas: Listas válidas de cualquier longitud y con cualquier combinación de valores en sus nodos ($\simeq \infty$)
- Entradas no válidas: Listas no válidas (null, listas no ordenadas, cíclicas, con compartición, etc) ($\simeq \infty$)
- Se debería quizás considerar la *precondición* y/o el *invariante de la representación* (o de la clase)

Testing de caja-negra

Ejemplo: Mezcla de listas enlazadas ordenadas

- Entradas válidas: Listas válidas de cualquier longitud y con cualquier combinación de valores en sus nodos ($\simeq \infty$)
- Entradas no válidas: Listas no válidas (null, listas no ordenadas, cíclicas, con compartición, etc) ($\simeq \infty$)
- Se debería quizás considerar la *precondición* y/o el *invariante de la representación* (o de la clase)

Ejemplo: Un compilador

- Habría que probar que los programas válidos (sintácticamente) compilan, pero también que los no válidos no compilan
- Entradas válidas: Conjunto de programas válidos ($\simeq \infty$)
- Entradas no válidas: Conjunto de programas no válidos ($\simeq \infty$)

Testing de caja-blanca

- En este caso se usa la estructura/lógica del programa (flujo de control y/o usos de variables)
- De nuevo, supongamos que queremos encontrar todos los errores:
  - Conjunto de casos que ejecuten todas las instrucciones? ⇒ Insuficiente

```
if (n > 0 && m < 0)
    x++;
...
```

  - Conjunto de casos que ejecuten todas las condiciones? ⇒ Insuficiente
  - Conjunto de casos que ejecuten todos los posibles caminos? ⇒ ...

Testing de caja-blanca

Si tenemos casos para todos los caminos, ¿estaríamos testeando completamente el programa? **La respuesta es no**, por dos razones:

1. Este conjunto es astronómico o infinito. Por ejemplo:

```
for (int i = 0; i < 20; i++){
    x = v[i];
    if (x < 20 || x > 50){
      ...
      if (x == 0) ...
    }
}
```

- El número de caminos en este caso es aproximadamente 10^{14}
- Si puedes escribir/ejecutar/verificar un caso cada 5 mins. $\Rightarrow$ 10^9 años.
- Si lo consigues en un seg. $\Rightarrow$ 3,2 millones de años
- Es cierto que hay caminos insatisfactibles pero sigue siendo imposible

Testing de caja-blanca

2. Incluso aunque se verificasen todos los caminos el programa podría tener errores. Hay dos explicaciones para esto:
   (a) Pueden faltar caminos. Por ej., puede faltar un camino `else if` en el anterior ejemplo.
   (b) Puede haber errores asociados a un camino que se produzcan o no dependiendo de los valores concretos de las variables

```
if (a-b < c)
   ...
```

Conclusión

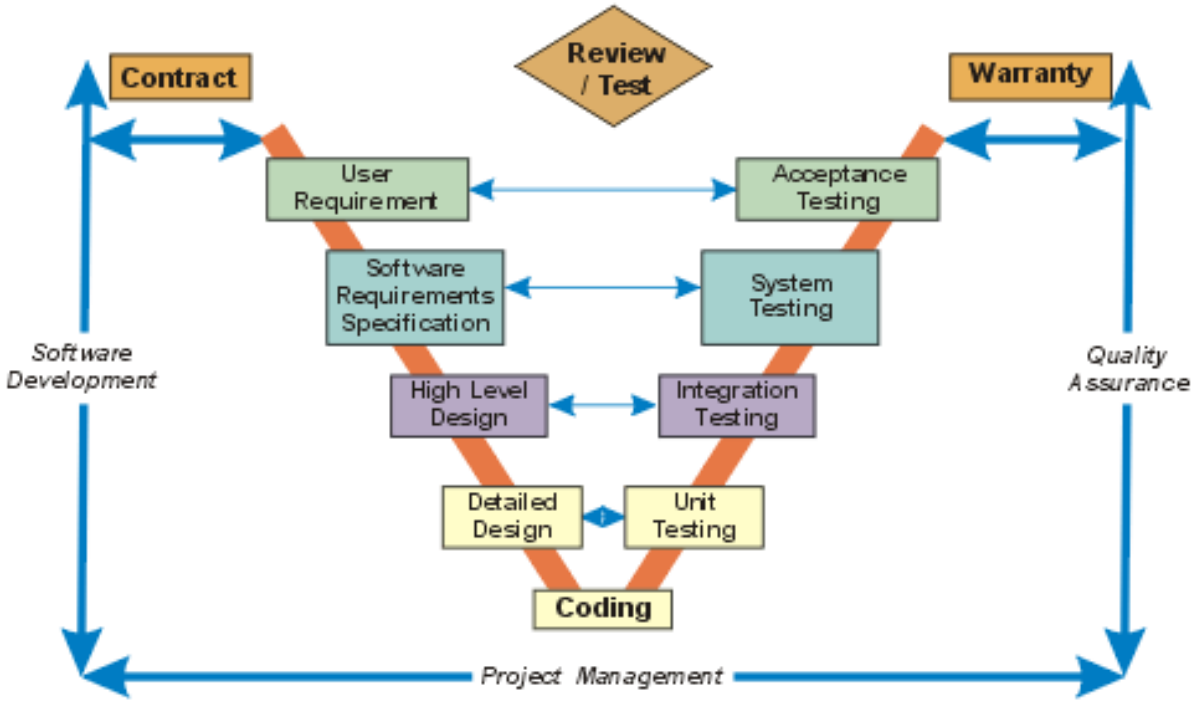
- Mediante testing dinámico no es posible asegurar que un programa no tiene errores, solamente es posible asegurar que sí los tiene.
- Cada enfoque tiene sus ventajas y desventajas.
  - Dependiendo del contexto será mejor usar uno u otro
  - En ocasiones lo más adecuado será combinarlos

Clases de testing de programas

- En función de si se ejecuta o no el programa:
  1. Testing estático ⇒ El programa no se ejecuta (ej. inspecciones, revisiones, verificación, análisis estático, . . .)
  2. Testing dinámico ⇒ El programa se ejecuta (ej. unit testing clásico)
- En función de qué información se usa:
  1. Testing de *caja-negra* (black-box testing) ⇒ Se usa la *especificación*
  2. Testing de *caja-blanca* (white-box testing) ⇒ Se usa el *código fuente*
- En función del nivel en el proceso de desarrollo de software:
  1. Testing de unidad (unit testing)
  2. Testing de integración (integration testing)
  3. Testing de sistema (system testing)
  4. Testing de aceptación (acceptance testing)

Niveles de testing

- La *SWEBOK* reconoce cuatro niveles de testing de software en función de la fase de desarrollo de software al que se asocian



Testing de unidad

- Se testea al nivel de la menor unidad funcional de un programa. Típicamente al nivel de procedimiento/función/método
- El nivel de *módulo* o *clase* en ocasiones también se considera testing de unidad
- Cuanto antes se encuentre un error menor es el coste de remediarlo
- Es posible paralelizar el proceso para diferentes unidades
- Dos consideraciones:
  1. Cómo diseñar casos de prueba adecuados
  2. En qué orden se deben realizar e integrar los distintos tests

Testing de integración

- Se trata de buscar errores en las interfaces y en la integración entre los diferentes componentes del programa
- La integración puede realizarse de forma incremental o no-incremental
- Si es incremental puede hacerse *top-down* o *bottom-up*

Testing de sistema

- El objetivo es probar el sistema completo para verificar que se cumplen los requisitos establecidos
- También debe verificarse que no se producen efectos no deseados en el entorno operativo del programa
- Se distinguen una serie de categorías (15 en ArtOfST):

1. Facility testing
2. Volume testing
3. Stress testing
4. Usability testing
5. Security testing
6. Performance testing
7. Storage testing
8. Configuration testing
9. Compatibility/Conversion testing
10. Installability testing
11. Reliability testing
12. Recovery testing
13. Serviceability testing
14. Documentation testing
15. (User) Procedure testing

Testing de aceptación

- Al igual que en el testing de sistema se prueba el sistema completo para verificar que se cumplen los requisitos establecidos
- La diferencia es que en este caso el cliente o usuario final realiza las pruebas, o al menos, participa en el proceso
- A.k.a *beta testing* o *end-user testing*
- Se usan casos de prueba (casos de uso)
- Idealmente, en el diseño de los casos, y en el proceso en general, colaboran: los clientes, los analistas, los *testers* y los desarrolladores.