PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 08-19

Análisis y Ejemplos de la Taxonomía de Defectos de Beizer

Diego Vallespir **Stephanie De León**

2008

Análisis y Ejemplos de la Taxonomía de Defectos de Beizer
Vallespir, Diego; De León Stephanie
ISSN 0797-6410
Reporte Técnico RT 08-19
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, 2008

Análisis y Ejemplos de la Taxonomía de Defectos de Beizer

Diego Vallespir
Laboratorio Cero Defectos
Grupo de Ingeniería de Software
Instituto de Comptuación
dvallesp@fing.edu.uy

Stephanie De León
Laboratorio Cero Defectos
Grupo de Ingeniería de Software
Instituto de Comptuación
sdeleon@fing.edu.uy

6 de diciembre de 2008

Abstract

Para establecer una estrategia óptima de remoción de defectos se debe conocer la efectividad y el costo de varios tipos de técnicas de remoción de defectos. Además, se debe conocer cómo varía la efectividad para cada tipo de defecto. Es decir, la técnica A1 puede ser muy buena para encontrar el tipo de defecto D1 pero no el D2.

Para conocer la efectividad por tipo de defecto es necesario contar con una clasificación de los tipos de defectos. Existen diversas taxonomías de defectos de software, aquí se presenta la taxonomía de Beizer. Esta taxonomía es de interés para nuestro trabajo, conocer la efectividad de las técnicas de verificación por tipo de defecto, por ser muy detallada y completa.

Beizer no explica la taxonomía en toda su extensión y existen partes de la misma que no son del todo claras. En este trabajo se busca recorrer la taxonomía y brindar explicaciones y ejemplos para cada uno de los tipos de defectos propuestos en la misma.

Índice

1. Introducción

Para establecer una estrategia óptima de remoción de defectos se debe conocer la efectividad y el costo de varios tipos de técnicas de remoción de defectos. Además, se debe conocer cómo varía la efectividad para cada tipo de defecto. Es decir, la técnica A1 puede ser muy buena para encontrar el tipo de defecto D1 pero no el D2.

Esta información se necesita para todas las fases de un proyecto de desarrollo de software: requerimientos, diseño, revisión de código, compilación y verificación. Para conseguir esta información se necesita contar con una taxonomía de defectos. Este trabajo presenta la taxonomía de defectos de Beizer [1].

La taxonomía de Beizer es muy grande y muy detallada en comparación con otras taxonomías, por ejemplo, comparada con la Clasificación Ortogonal de Defectos de IBM [2]. El nivel de detalle de la taxonomía puede ser interesante para conocer el rendimiento de las técnicas de remoción de defectos para cada tipo de defecto. Sin embargo, entendemos que esta taxonomía, por el mismo nivel de detalle, puede ser muy difícil de utilizar en la práctica.

Beizer no explica la taxonomía en toda su extensión y existen partes de la misma que no son del todo claras. En este trabajo se busca recorrer la taxonomía y brindar explicaciones y ejemplos para cada uno de los tipos de defectos propuestos en la misma.

De forma de distinguir nuestro trabajo de la taxonomía en sí misma presentamos cada categoría de defectos de la siguiente forma:

- **Nombre del tipo de defecto** (categorías y subcategorías).
- **Descripción del tipo de defecto.** Se presenta la descripción dada por Beizer en su idioma original.
- **Complemento para el tipo de defecto.** Cuando entendemos que la definición del tipo de defecto provista por Beizer no es del todo clara se agrega un complemento. El mismo se escribe en idioma español.
- textbfEjemplos de defectos para cada categoría. Se presentan algunos ejemplos para las categorías. Estos ejemplos son también de creación propia.

La taxonomía es jerárquica. Esto quiere decir que un tipo de defecto se puede dividir en varios subtipos y así sucesivamente. Para reflejar esto cada tipo de defecto tiene un número de 4 dígitos. A veces se usan subnúmeros, en ese caso se usan puntos, por ejemplo: “1234.1.6”. La letra “x” se usa como un marcador de posición para un futuro llenado con números a medida que la taxonomía se expande. Por ejemplo:

```
3xxx - structural bugs in the implemented software
  32xx - processing bugs
    322x - expression evaluation
    3222 - arithmetic expressions
        3222.1 - wrong operator
```

El último dígito de un conjunto es siempre el 9, por ejemplo: 9xxx, 39xx, 3229. Estas categorías se usan cuando no está disponible una descomposición más fina y el defecto encontrado no coincide exactamente con una de las categorías. Por ejemplo, un defecto no clasificado es un defecto 9xxx, un defecto estructural en el software implementado (3xxx) que no está clasificado es 39xx.

Los grupos principales de la taxonomía son los siguientes 9:
1xxx - FUNCTIONAL BUGS: REQUIREMENTS AND FEATURES
2xxx - FUNCTIONALITY AS IMPLEMENTED
3xxx - STRUCTURAL BUGS
4xxx - DATA
5xxx - IMPLEMENTATION
6xxx - INTEGRATION
7xxx - SYSTEM AND SOFTWARE ARCHITECTURE
8xxx - TEST DEFINITION OR EXECUTION BUGS
9xxx - OTHER BUGS, UNSPECIFIED

Nuestra investigación sobre la efectividad de las técnicas de verificación se restringe a la verificación unitaria. Esta restricción implica que las categorías 1xxx, 7xxx y 8xxx no sean analizadas porque escapan a nuestro trabajo. La categoría 1xxx trata de defectos en los Requerimientos y estos son considerados como correctos en nuestro trabajo. La categoría 7xxx trata defectos en la arquitectura de software y/o sistema, este tipo de defectos tampoco es relevante para nuestro estudio. Por último, la categoría 8xxx considera los defectos en las propias pruebas de software y esto queda también por fuera de nuestra investigación.

2. FUNCTIONALITY AS IMPLEMENTED - 2xxx

La siguiente definición, extraída de Beizer [1], es necesaria para la comprensión de la categoría:
feature: *(ANSI87B) A software characteristic specified or implied by requirements documentation, such as functionality, performance, design constraints, or behavioral constraints.*

A lo largo del documento se utiliza el término *característica* para referir al término *feature* definido por Beizer.

Definición: Requirement known or assumed to be correct, implementable, and testable, but implementation is wrong.

Complemento: Esta categoría trata el error a *alto nivel*. Los requerimientos se asumen correctos. Abarca los siguientes casos:

- Diseño erróneo. El diseño no implementa la característica (feature) de forma correcta.
- Implementación errónea. El diseño es correcto pero la implementación no cumple con la característica (feature) de forma correcta.

En un nivel mas fino, la categoria se divide en:

- Correctness (21xx)
- Completeness, Features (22xx)
- Completeness, Cases (23xx)
- Domains(24xx)
- User Messages and Diagnostics (25xx)
- Exception Conditions Mishandled (26xx)

Para clasificar en (21xx) debe existir la implementación de la característica. La correctitud es tomada solamente desde el punto de vista de una función matemática con entradas y salidas conocidas. Entonces, no entran aqui casos de completitud de la característica, como por ejemplo una característica no implementada. Este tipo de defectos corresponde a (22xx). Además, en (21xx) el defecto está en la característica en si, y no en casos particulares de la misma. Si el defecto es sobre completitud en los casos se refiere a (23xx).

(24xx) clasifica los errores de dominio, esto refiere a errores en el tratamiento de los datos de entrada, pero el procesamiento se considera correcto. Otras categorías registran defectos de mensaje y diagnóstico (25xx) y defectos por un manejo inapropiado de excepciones (26xx).

2.1. CORRECTNESS - 21xx

Definición: Having to do with the correctness of the implementation.

2.1.1. Feature misunderstood, wrong - 211x

Definición: Feature as implemented is not correct - not as specified.

Complemento: La implementación incorrecta de una característica puede ocurrir por dos razones:

Misunderstood: La implementación de la característica es incorrecta porque el diseñador/implementador entendió mal la especificación de la característica.

Wrong: Contraria a la anterior, la característica fue entendida de forma correcta por el diseñador/implementador, pero su implementación es incorrecta.

Ejemplo:

1. **misunderstood:** La especificación de requerimientos de una característica para una aplicación contable indica la forma de cálculo de un valor crítico para la aplicación. Un defecto de "Feature misunderstood, wrong" debido a un "mal entendimiento" es que el diseñador/implementador entienda mal el cálculo, y por consiguiente, implemente la característica de forma equivocada.

2. **wrong:** La especificación de requerimientos para el alta de empleados de un centro hotelero, indica que no puede haber más de un empleado con rol Gerente. El diseñador/implementador entiende correctamente la especificación de la característica, o sea, diseña/codifica pensando que el rol Gerente debe ser único. Pero al momento de diseñar/codificar olvida explicitar el chequeo que evita que haya mas de un empleado con rol Gerente, provocando que la característica esté mal implementada. El defecto clasifica aquí porque la característica alta de Gerente esta mal implementada como tal.

2.1.2. Feature interactions - 218x

Definición: Feature is correctly implemented by itself, but has incorrect interactions with other features, or specified or implied interaction is incorrectly handled.

Complemento: Toda aplicación tiene su conjunto particular de características y un conjunto mucho más grande de potenciales interacciones no especificadas entre características y por lo tanto de defectos en tales interacciones.

Ejemplo: Un teléfono cuenta con llamada en espera y reenvío de llamada. La llamada en espera permite poner una nueva llamada en espera mientras se continúa hablando con la primer llamada. El reenvío de llamada permite redireccionar llamadas entrantes a algún otro número de teléfono. Posibles interacciones entre características que pueden presentar defectos:

- ¿Se pone en espera una tercera llamada cuando ya hay una llamada en espera?
- ¿Se permite reenvío de una llamada reenviada?
- ¿Loop de reenvío de llamada?
- ¿Poner una llamada en espera mientras esta activo el reenvío de una llamada?
- ¿Iniciar un reenvío cuando hay una llamada en espera?
- ¿Poner en espera llamadas reenviadas cuando el teléfono al que son reenviadas (no) tiene reenvío de llamada?

2.2. COMPLETENESS, FEATURES - 22xx

Definición: Having to do with the completeness with which features are implemented.

2.2.1. Missing feature - 221x

Definición: An entire feature is missing.

Ejemplo: La especificación de requerimientos para un sistema de cobro incluye la característica de impresión de un comprobante. Esta característica no se incluye en la implementación del sistema.

2.2.2. Unspecified feature - 222x

Definición: A feature not specified has been implemented.

Ejemplo: El caso inverso al anterior: el sistema permite imprimir la liquidación, pero esta característica no forma parte de la especificación de requerimientos.

2.2.3. Duplicated, overlapped feature - 223x

Definición: Feature as implemented duplicates or overlaps features implemented by other parts of the software.

Ejemplo: Un supermercado con sucursales tiene dos tipos de sistemas. Un sistema para la casa central (sistema central) y otro para las sucursales (sistema sucursal). En ambos sistemas se tiene implementado de forma diferente el alta de usuario. La característica está duplicada ya que se podría haber usado el alta de uno de los sistemas en el otro.

2.3. COMPLETENESS, CASES - 23xx

Definición: Having to do with the completeness of cases within features.

2.3.1. Missing case - 231x

Definición: An entire case is missing.

Ejemplo: La especificación de requerimientos de una característica que implica la impresión de un documento, indica que se puede realizar aunque el documento esté incompleto. El diseño y/o la implementación del sistema no consideran el caso de imprimir el documento cuando está incompleto. Sin embargo, sí se puede imprimir el documento cuando está finalizado. Por esto último es que este defecto se califica en (231x) y no en (221x) (es un caso no implementado de una característica, pero la característica está implementada).

2.3.2. Extra case - 232x

Definición: Cases which should not have been handled are.

Ejemplo: El caso inverso al anterior: el sistema permite imprimir la liquidación cuando está incompleta y cuando está completa, cuando en la especificación de requerimientos se especifica sólo la impresión del documento completo.

2.3.3. Duplicated, overlapped case - 233x

Definición: Duplicated handling of cases or partial overlap with other cases.

Ejemplo: Idem (223x), aplicado en vez de a una característica, a uno de los casos de la misma.

2.3.4. Extraneous output data - 234x

Definición: Data not required is output.

Ejemplo: La especificación de requerimientos de un sistema de cobro incluye la funcionalidad de impresión de comprobantes de pago. Hay dos tipos de impresión, una del comprobante en detalle, y otra sólo con los montos (son dos casos distintos de la característica imprimir comprobante). En la impresión del comprobante en detalle (primer caso) aparecen el tipo de hoja, y datos sobre el formato de impresión utilizados. Estos datos no se requerían en la especificación de requerimientos para el caso de impresión de comprobante detallada.

2.4. DOMAINS - 24xx

Definición: Processing case or feature depends on a combination of input values. A domain bug exists if the wrong processing is executed for the selected input-value combination.

The bug assumption for domain testing is that processing is okay but the domain definition is wrong.

2.4.1. Domain misunderstood, wrong - 241x

Definición: Misunderstanding of the size, shape, boundaries, or other characteristics of the specified input domain for the feature or case. Most bugs related to handling extreme cases are domain bugs.

Ejemplo: En la especificación de requerimientos de un sistema medico, se pide que se traten de forma distinta los datos correspondientes a las personas mayores de edad de las personas menores de edad. La especificación indica que las personas mayores de edad son aquellas que tienen o han superado los 18 años. Por alguna razón el diseñador/implementador pasa por alto este último punto y al implementar el dominio, lo hace considerando a personas mayores de edad como aquellas con 21 años o mas. Este defecto es causa de no haber entendido el dominio.

2.4.2. Boundary locations - 242x

Definición: The values or expressions which define a domain boundary are wrong: e.g., "X>=6" instead of "X>=3".

2.4.3. Boundary closures - 243x

Definición: End points and boundaries of the domain are incorrectly associated with an adjacent domain: e.g., "X>=0" instead of "X>0".

2.4.4. Boundary intersections - 244x

Definición: Domain boundaries are defined by a relation between domain control variables. That relation, as implemented, is incorrect: e.g., "IF X>0 AND Y>0 . . . " instead of "IF X>0 OR Y>0 . . . ".

2.5. USER MESSAGES AND DIAGNOSTICS - 25xx

Definición: User prompt or printout or other form of communication is incorrect. Processing is assumed to be correct: e.g., a false warning, failure to warn, wrong message, spelling, formats.

Ejemplo: Mensaje en tono inapropiado dirigido al usuario. Por ejemplo, cuando va a cerrar un documento, el sistema pregunta con un mensaje informal si desea guardar antes de cerrar.

2.6. EXCEPTION CONDITIONS MISHANDLED - 26xx

Definición: Exception conditions such as illogicals, resource problems, failure modes, which require special handling, are not correctly handled or the wrong exception-handling mechanisms are used.

Ejemplo: La cola de impresión tiene un problema y manda un mensaje al sistema. Esta situación anormal no es manejada de forma correcta por el mismo y el sistema termina abruptamente su ejecución.

3. STRUCTURAL BUGS - 3xxx

Definición: Bugs related to the component's structure: i.e., the code.

Complemento: Structural Bugs comprende los defectos en la estructura del código, como ser los flujos de control (31xx) y el procesamiento (32xx). Quedan fuera los defectos de datos, ya sea inicialización o manipulación de los mismos. Estos defectos entran en la categoria (4xxx).

3.1. CONTROL FLOW AND SEQUENCING - 31xx

Definición: Bugs specifically related to the control flow of the program or the order and extent to which things are done, as distinct from what is done.

3.1.1. General structure - 311x

Definición: General bugs related to component structure.

3.1.1.1 Unachievable path - 3112

Definición: A functionally meaningful processing path in the code for which there is no combination of input values which will force that path to be executed. Do not confuse with unreachable code. The code in question might be reached by some other path.

Complemento: Un camino inaccesible en el código se vuelve un defecto cuando no poder recorrerlo hace que no se ejecute el procesamiento funcionalmente significativo ligado al mismo. Esta aclaración es válida ya que en la lectura de la definición suele pasarse por alto la primer parte: "A functionally meaningful processing path".

Ejemplo:

```
if ( a > 1 and b = 0 ) {   ( A )
   x = 0                   ( B )
}
if( x > 1 ) {              ( C )
   x = x + 1               ( D )
}
```

No hay conjunto de datos de entrada que logre que se ejecute el camino ABCD.

3.1.1.2 Unreachable code - 3114

Definición: Code for which there is no combination of input values which will cause that code to be executed.

Ejemplo:

```
...
a = b
if ( a ! = b ){   /* nunca se va a hacer verdadera la condición */
   ...
   ...
   ...
}
```

3.1.1.3 Dead-end code - 3116

Definición: Code segments which once entered cannot be exited, even though it was intended that an exit be possible.

Ejemplo:

```
...
i = 1;
while(i > 0){
   ...
   i = i + 1;   /* en vez de poner i = i - 1 */
   ...
}
...
```

Se supone que la variable i no es modificada dentro del while, fuera de la línea en que se incrementa.

3.1.2. Control logic and predicates - 312x

Definición: The path taken through a program is directed by control flow predicates (e.g., boolean expressions). This category addresses the implementation of such predicates.

3.1.2.1 Duplicated logic - 3122

Definición: Control logic which should appear only once is inadvertently duplicated in whole or in part.
Ejemplo:

```
...
if (persona.peso() > sobrepeso) {
   processing1
}
else{
   processing3
}
if (persona.peso() > sobrepeso) {
   processing2
}
else{
   processing4
}
...
```

En este ejemplo existe duplicación porque *processing2* y *processing4* deberían pertenecer al mismo bloque del if-then-else que *processing1* y *processing3* respectivamente. En cambio se duplica el predicado y se colocan por separado, en los respectivos bloques. Esto puede ocasionar una falla en un futuro, por ejemplo realizar mantenimiento o correcciones, en caso de modificar la condición, puede ocurrir que se modifique la primera y no la segunda, dando lugar a inconsistencias.

El ejemplo se plantea bajo el supuesto que ninguno de los "processing" modifica *persona.edad()*. Sino pasaría a ser un defecto de correctitud de implementación de la funcionalidad (211x).

Este es un ejemplo de lógica repetida en parte, los casos en los que se repite el bloque entero (no solo la condición) son defectos que también caen en esta categoría.

3.1.2.2 Don't care - 3124

Definición: Improper handling of cases for which what is to be done does not matter either because the case is impossible or because it really doesn't matter: e.g., incorrectly assuming that the case is a don't-care case, failure to do case validation, not invoking the correct exception handler, improper logic simplification to take advantage of such cases.

Complemento: En esta subcategoría entran los defectos que ocurren debido a un manejo inapropiado de casos para los cuales no importa lo que se hace, ya sea porque el caso es imposible o porque realmente no importa. Asumir que un caso no importa cuando en realidad sí, fallar al realizar la validación de un caso que "no importa", invocar el manejador de excepción incorrecto, realizar una simplificación lógica inapropiada tomando ventaja de que el caso "no importa" son todos defectos que caen en esta categoría.

3.1.2.3 Illogicals - 3126

Definición: Improper identification of, or processing of, illogical or impossible conditions. An illogical is stronger than a don't care. Illogicals usually mean that something bad has happened and that recovery is needed.

Examples of bugs include: illogical not really so, failure to recognize illogical, invoking wrong handler, improper simplification of control logic to take advantage of the case.

Complemento: En esta subcategoría entran los defectos relacionados con la identificación o el procesamiento inapropiado de condiciones ilógicas o imposibles. Esta subcategoría es más fuerte que la anterior (3124).

3.1.2.4 Other control flow predicate bugs - 3128

Definición: Control-flow problems which can be directly attributed to the incorrect formulation of a control flow predicate: e.g., "IF A>B THEN..." instead of "IF A<B THEN...".

3.1.3. Case selection bug - 313x

Definición: Simple bugs in case selections, such as improperly formulated case selection expression, GOTO list, or bug in assigned GOTO.

3.1.4. Loops and iteration - 314x

Definición: Bugs having to do with the control of loops.

3.1.4.1 Initial value - 3141

Definición: Initial iteration value wrong: e.g., "FOR I = 3 TO 17..." instead of "FOR I = 8 TO 17...".

3.1.4.2 Terminal value or condition - 3142

Definición: Value, variable, or expression used to control loop termination is incorrect: e.g., "FOR I = 3 TO 17..." instead of "FOR I = 3 TO 10...".

3.1.4.3 Increment value - 3143

Definición: Value, variable, or expression used to control loop increment value is incorrect: e.g., "FOR I = 1 TO 7 STEP 2..." instead of "FOR I = 1 TO 7 STEP 5...".

3.1.4.4 Iteration variable processing - 3144

Definición: Where end points and/or increments are controlled by values calculated within the loop's scope, a bug in such calculations.

3.1.4.5 Exception exit condition - 3148

Definición: Where specified values or conditions or relations between variables force an **abnormal** exit to the loop, either incorrect processing of such conditions or incorrect exit mechanism invoked.

Complemento: No confundir con Exception Conditions Mishandled (26xx), en donde clasifican los defectos en el manejo de excepciones a nivel de las características, no a nivel de loops.

3.1.5. Control initialization and/or state - 315x

Definición: Bugs having to do with how the program's control flow is initialized and changes of state which affect the control flow: e.g., switches.

3.1.5.1 Control initialization - 3152

Definición: Initializing to the wrong state or failing to initialize.

Ejemplo:

```
const INI = ...;
...
A = inicializar();
if(A < INI) GOTO X
else GOTO Y;
```

Los defectos posibles del código presentado que entran en esta subcategoría son:

1. Que falle la función *inicializar()* ("failing to initialize").
2. Que se inicialice el flujo de control en un estado incorrecto, esto ocurre si X o Y no son las etiquetas (estados) a las que deberían ir los GOTO's.

3.1.5.2 Control state - 3154

Definición: For state determined control flows, incorrect transition to a new state from the current state: e.g., input condition X requires a transition to state B, given that the program is in state A, instead, the transition is to state C. Most incorrect GOTOs are included in this category.

Ejemplo:

```
...
IF X THEN GOTO C;
...
```

Para este ejemplo suponemos que si evaluar X da True el GOTO debería ir a la etiqueta B, en vez de ir a la C.

3.1.6. Incorrect exception handling - 316x

Definición: Any incorrect invocation of a control-flow exception handler not previously categorized.

Complemento: Es necesario hacer la distinción entre esta clasificación y Exception Conditions Mishandled(26xx) y Exception exit condition (3148). La categoría (26xx) trata los defectos a un nivel mayor, a nivel de característica. Esta categoría (316x) considera defectos en excepciones en flujos de control excluyendo los defectos de manejo de excepciones dentro de loops e iteraciones, que clasifican en (3148).

3.2. PROCESSING - 32xx

Definición: Bugs related to processing under the assumption that the control flow is correct.

3.2.1. Algorithmic, fundamental - 321x

Definición: Inappropriate or incorrect algorithm selected, but implemented correctly: e.g., using an incorrect approximation, using a shortcut string search algorithm that assumes string characteristics which may not apply.

Ejemplo: Error de aproximación, lo correcto sería aplicar redondeo a determinado real, dejándolo con 2 decimales, y el sistema lo trunca, dejándolo entero.

3.2.2. Expression evaluation - 322x

Definición: Bugs having to do with the way arithmetic, boolean, string, and other expressions are evaluated.

3.2.2.1 Arithmetic - 3222

Definición: Bugs related to evaluation of arithmetic expressions.

3.2.2.1.1 Operator - 3222.1

Definición: Wrong arithmetic operator or function used.

Ejemplo: Usar predecesor en vez de sucesor.

3.2.2.1.2 Parentheses - 3222.2

Definición: Syntactically correct bug in placement of parentheses or other arithmetic delimiters.

Ejemplo: Escribir la expresión aritmética (2 / 4) + 5, en vez de 2 / (4 + 5). En ambos casos el uso de los paréntesis es sintácticamente correcto, están balanceados. El error es a nivel semántico, ocurre en la evaluación de la expresión.

3.2.2.1.3 Sign - 3222.3

Definición: Bug in use of sign.

Ejemplo: Darle a una variable valor negativo, cuando debería ser positivo, o viceversa.

3.2.2.2 Logical or boolean, not control - 3224

Definición: Bug in the manipulation or evaluation of boolean expressions which are not (directly) part of control-flow predicates: e.g., using wrong mask, AND instead of OR, incorrect simplification of boolean function.

3.2.2.3 String manipulation - 3226

Definición: Bug in string manipulation.

3.2.2.3.1 Beheading - 3226.1

Definición: The beginning of a string is cut off when it should not have been or not cut off when it should be.

3.2.2.3.2 Curtailing - 3226.2

Definición: As for beheading but for string end.

3.2.2.3.3 Concatenation order - 3226.3

Definición: Strings are concatenated in wrong order or concatenated when they should not be.

- Append instead of precede - 3226.3.1.
- Precede instead of append - 3226.3.2.

3.2.2.3.4 Inserting - 3226.4

Definición: Having to do with the insertion of one string into another.

Ejemplo: Error en la posición en la que se inserta un string en otro.

3.2.2.3.5 Converting case - 3226.5

Definición: Case conversion (upper to lower, say) is incorrect.

3.2.2.3.6 Code conversion - 3226.6

Definición: String is converted to another code incorrectly or not converted when it should be.

Complemento: En esta categoría no clasifican los defectos de transformación de tipo String a otros tipos, como por ejemplo a Integer, Float, etc. Estos defectos corresponden a la categoría (4214) Type transformation.

3.2.2.3.7 Packing, unpacking - 3226.7

Definición: Strings are incorrectly packed or unpacked.

3.2.2.4 Symbolic, algebraic - 3228

Definición: Bugs in symbolic processing of algebraic expressions.

Complemento: Los defectos que entran en esta categoría refieren a los que ocurren en el procesamiento de expresiones aritméticas con símbolos.

Ejemplo: Al evaluar A + A + A + B, se evalúa 2A + B, en vez de 3A + B.

3.2.3. Initialization - 323x

Definición: Bugs in initialization of variables, expressions, functions, etc. used in processing, excluding initialization bugs associated with declarations and data statements and loop initialization.

Complemento: Esta categoría trata los defectos de inicialización de variables utilizadas en procesamiento. Son variables auxiliares, luego de su uso local no tienen importancia. Esta inicialización excluye la inicialización asociada a la declaración de datos relevantes al programa (413x) Initial, default values y la inicialización de loops (3141)Initial value.

Ejemplo: En el siguiente código el defecto está en la inicialización de la variable “min”. Observar que esta variable es auxiliar, utilizada para hallar el índice que contiene el mínimo entero del vector. Este índice si es una variable relevante a efectos del sistema. Si “min” fuera relevante, un error en su inicialización no clasificaría aquí, sino en Initial, default values(413x). “min” tampoco es la variable sobre la que itera el loop, por lo que su inicialización no se incluye en lo que es la inicialización del loop.

```
min = MIN_INT   //en vez de inicializarse con el máximo valor entero,
                //se inicializa con un valor negativo
for ( i=0; i < vector.size(); i++ ){
   if ( vector[i] < min ){
      min=vector[i]
      indice=i;
   }
   i++;
}
return i;
```

3.2.4. Cleanup - 324x

Definición: Incorrect handling of cleanup of temporary data areas, registers, states, etc. associated with processing.

3.2.5. Precision, accuracy - 325x

Definición: Insufficient or excessive precision, insufficient accuracy and other bugs related to number representation system used.

Ejemplo: Expresar un real con cantidad de decimales de mas o de menos (precisión).

3.2.6. Execution time - 326x

Definición: Excessive (usually) execution time for processing component.

Ejemplo: La implementación del componente enlentece el tiempo de ejecución excesivamente. Cambiando el código de manera de optimizarlo se podría bajar esta demora.

4. DATA - 4xxx

Definición: Bugs in the definition, structure, or use of data.

4.1. DATA DEFINITION, STRUCTURE, DECLARATION - 41xx

Definición: Bugs in the definition, structure and initialization of data: e.g., in DATA statements. This category applies whether the object is declared statically in source code or created dynamically.

4.1.1. Type - 411x

Definición: The data object type, as declared, is incorrect: e.g., integer instead of floating, short instead of long, pointer instead of integer, array instead of scalar, incorrect user-defined type.

4.1.2. Dimension - 412x

Definición: For arrays and other objects which have a dimension (e.g., arrays, records, files) by which component objects can be indexed, a bug in the dimension or in the minimum or maximum dimensions, or in redimensioning statements.

Ejemplo:

```
int array[10];
```

en vez de:

```
int array[100];
```

4.1.3. Initial, default values - 413x

Definición: Bugs in the assigned initial values of the object (e.g., in DATA statements), selection of incorrect default values, or failure to supply a default value if needed.

Complemento: Diferenciar los defectos de inicialización de variables de "datos" de las que se utilizan localmente de forma auxiliar para realizar determinado procesamiento. Defectos en la inicialización de estas últimas clasifican en (323x) Initialization.

Ejemplo: El siguiente ejemplo muestra un defecto en el valor inicial asignado al array.

```
int[] cadena = {4,7,17} //en realidad deberia ser {5,7,17}
```

4.1.4. Duplication and aliases - 414x

Definición: Bugs related to the incorrect duplication or failure to create a duplicated object.

4.1.4.1 Duplicated - 4142

Definición: Duplicated definition of an object where allowed by the syntax.

Ejemplo: En un mismo scope de un programa se permite definir mas de una variable con el mismo nombre, esto puede traer inconsistencias y perdidas de información.

```
...
int x;
...
int x;
```

4.1.4.2 Aliases - 4144

Definición: Object is known by one or more aliases but specified alias is incorrect; object not aliased when it should have been.

Ejemplo: No se crea el alias de un objeto cuando se debió haber creado uno. La siguiente es una función que recibe un entero y lo coloca al final de una lista. La lista es global al programa. Se supone que la lista tiene al menos un elemento.

```
void ordenar(int n){//prt_lista es el puntero al comienzo
                    //de la lista global
   while(ptr_lista.next() \section{= NULL){
      ptr_lista = prt_lista.next();
   }
   ptr_lista.next().agregar(n);
}
```

El error del código anterior es que al recorrer la lista con el puntero que apunta al principio de la misma, se pierde la referencia al principio de la lista. Lo correcto hubiera sido crear un alias de ptr_lista y recorrer la lista con este alias. De esta manera no se pierde al referencia al principio de la lista.

```
void ordenar(int n){//prt_lista es el puntero al comienzo
                    //de la lista global
   lista * ptr_aux = ptr_lista;
   while(ptr_aux.next() \section{= NULL){
      ptr_aux = prt_aux.next();
   }
   ptr_aux.next().agregar(n);
}
```

4.1.5. Scope - 415x

Definición: The scope, partition, or components to which the object applies is incorrectly specified.

4.1.5.1 Local should be global - 4152

Definición: A locally defined object (e.g., within the scope of a specific component) should have been specified more globally (e.g., in COMMON).

Ejemplo: Es de esperar que el siguiente programa de un error en tiempo de compilación dado que la variable *x* fue declarada dentro del scope del while, por lo tanto, fuera del mismo no existe. El error esta en que se utiliza después que termina el while. Debió haber sido declarada antes de la declaración del while.

```
...
while (condicion){
   int x;
   x = function();
}
System.out.println("El resultado es: " + x);
...
```

4.1.5.2 Global should be local - 4154

Definición: The scope of an object is too global, it should have been declared more locally.

Ejemplo: En el siguiente código la variable y fue declarada globalmente innecesariamente dado que solo se utiliza dentro de la función fun. Lo mas correcto es declararla dentro de la función.

```
...
int y;
int func(){
   ...
   y = ...;
   ...
   return y;
}
```

4.1.5.3 Global/local inconsistency or conflict - 4156

Definición: A syntactically acceptable conflict between a local and global declaration of an object (e.g., incorrect COMMON).

Ejemplo:

```
int x;
x = func1();
int func2(){
   int x;
   x = fun_aux();
   ...
   x = x + x; //una de estas x tendria que ser la x declarada
              //globalmente para ello se debería haber nombrado
              //distinto a una de las dos x.
```

4.1.6. Static/dynamic resources - 416x

Definición: Related to the declaration of static and dynamically allocated resources.

Complemento: Unos de los casos mas claros de declaración estática y dinámica de un recurso es el de la memoria principal.

4.1.6.1 Should be static resource - 4162

Definición: Resource is defined as a dynamically allocated object but should have been static (e.g., permanent).

Ejemplo: Se define un array con tamaño dependiente de una variable que toma valor en tiempo de ejecución, cuando en realidad su tamaño debería ser estático (valor en tiempo de compilación).

```
int array [n];
```

en vez de:

```
int array [5]; // n es una variable que toma valor antes de la
               //declaración dinámica del array
```

4.1.6.2 Should be dynamic resource - 4164

Definición: Resource is defined as static but should have declared as dynamic.

Ejemplo: Se define un array con tamaño fijo cuando en realidad el tamaño se tendría que definir en tiempo de ejecución.

```
int array [5];
```

en vez de:

```
int array [n]; // n es una variable que toma valor antes de la
               //declaración dinámica del array
```

4.1.6.3 Insufficient resources, space - 4166

Definición: Number of specified resources is insufficient or there is insufficient space (e.g., main memory, cache, registers, disc, etc.) to hold the declared resources.

Complemento: Refiere a un dump en la memoria, generalmente manifestado por un "segmentation fault". Esto ocurre cuando se quiere acceder a un espacio de memoria que no fue asignado, o cuando se desea reservar más memoria que la disponible. Dicha reserva puede ser dinámica o estática.

4.1.6.4 Data overlay bug - 4168

Definición: Data objects are to be overlayed but there is a bug in the specification of the overlay areas.

Ejemplo: Contamos con dos estructuras de datos: un árbol binario, y un vector. El árbol contiene datos solo en sus hojas. Cada elemento del vector es un puntero a una hoja del árbol. De esto surge un área de datos superpuesta, formada por el conjunto de hojas del árbol. Dicha información es compartida entre el árbol y el vector. Un defecto en la especificación del área de superposición de datos es que los elementos del vector hagan referencia a las hojas equivocadas, o que se referencien hojas de más o de menos, o que los datos de las hojas sean incorrectos.

4.2. DATA ACCESS AND HANDLING - 42xx

Definición: Having to do with access and manipulation of data objects that are presumed to be correctly defined.

4.2.1. Type - 421x

Definición: Bugs having to do with the object type.

4.2.1.1 Wrong type - 4212

Definición: Object type is incorrect for required processing: e.g., multiplying two strings.

4.2.1.2 Type transformation - 4214

Definición: Object undergoes incorrect type transformation: e.g., integer to floating, pointer to integer, specified type transformation is not allowed, required type transformation not done. Note, type transformation bugs can exist in any language, whether or not it is strongly typed, whether or not there are user-defined types.

4.2.1.3 Scaling, units - 4216

Definición: Scaling or units (semantic) associated with object is incorrect, incorrectly transformed, or not transformed: e.g., FOOT POUNDS to STONE FURLONGS.

4.2.2. Dimension - 422x

Definición: For dynamically variable dimensions of a dimensioned object, a bug in the dimension: e.g., dynamic redimension of arrays, exceeding maximum file length, removing one more than the minimum number of records.

4.2.3. Value - 423x

Definición: Having to do with the value of data objects or parts thereof.

4.2.3.1 Initialization - 4232

Definición: Initialization or default value of object is incorrect. Not to be confused with initialization and default bugs in declarations. This is a dynamic initialization bug.

Complemento: Esta subcategoría trata los defectos generados por un error en el valor inicial o default asignado a un objeto. Dicho asignación se realiza dinámicamente (el valor inicial o por defecto asignado en tiempo de compilacion), esto es lo que diferencia a esta categoría de la (413x) – Inicial, default values.

Ejemplo: Carga dinámica de los valores de un array. El defecto en este ejemplo esta en el valor inicial asignado al array, dicho valor se asigna dinámicamente.

```
int valor_ini = valor_inicial();
for (int i = 0; i < n; i++){   // n es el tamaño del vector
   vector[i] = valor_ini;
}
```

4.2.3.2 Constant value - 4234

Definición: Incorrect constant value for an object: e.g., a constant in an expression.

Complemento: El valor constante al que hace referencia la definición no debe considerarse sólo como la declaración de una constante, cuyo nombre se utiliza en una expresión, sino también como los valores constantes que aparecen en el programa. Por ejemplo: el número 5, el carácter 'a', el string "hola", etc.

Ejemplo: Se asigna un valor incorrecto a una constante que es utilizada mas adelante en el programa.

```
const fiebre = 38; // se debió asignar 37 en vez de 38
...
if (temperatura > fiebre){
...
}
```

El ejemplo presentando podría dar lugar a confusión al momento de clasificar el defecto. A simple vista, parece que podría clasificar en Domain misunderstood, wrong (241x); Boundary locations (242x) o en Other control flow predicate bugs (3128).

Suponiendo que *fiebre* no es parte del dominio, este ejemplo no clasifica en (241x) ni en (242x).

Aparte, el defecto no clasifica en (3128) porque no se considera que el predicado esté mal formulado, sino que el valor de la constante no es el correcto.

4.2.4. Duplication and aliases - 424x

Definición: Bugs in dynamic (run time) duplication and aliasing of objects.

4.2.4.1 Object already exists - 4242

Definición: Attempt to create an object which already exists.

Complemento: Esta subcategoría se asemeja a (4142) – Duplicated. La diferencia entre ambas es que en la subcategoría Duplicated la duplicación de los objetos es estática, se hace en tiempo de compilación. En cambio, en esta subcategoría se intenta duplicar un objeto de forma dinámica, o sea en tiempo de ejecución, por lo que cuando se va a realizar la duplicación uno de los objetos ya existe.

Ejemplo:

```
...
if(cond1){
   int x = ...;
   ...
}
else{
   ...
}
if(cond2){
   int x = ...;
   ...
}
else{
   ...
}
```

Si *cond1* y *cond2* se evalúan verdaderas, resulta en la doble definición de *x*.

4.2.4.2 No such object - 4244

Definición: Attempted reference to an object which does not exist.

Complemento: No confundir con (6111) No such component. Se diferencian en el concepto al que se refieren. (4244) refiere a "objects", y (6111) a "components". El concepto de (6111) esta en un nivel mas alto, mas general, que (4244).

4.2.5. Resources - 426x

Definición: Having to do with dynamically allocated resources and resource pools, in whatever memory media they exist: main, cache, disc, bulk RAM. Included are: queue blocks, control blocks, buffer blocks, heaps, files, etc.

4.2.5.1 No such resource - 4262

Definición: Referenced resource does not exist.

4.2.5.2 Wrong resource type - 4264

Definición: Wrong resource type referenced.

4.2.6. Access - 428x

Definición: Having to do with the access of objects as distinct from the manipulation of objects. In this context, accesses include read, write, modify, and (in some instances) create and destroy.

4.2.6.1 Wrong object accessed - 4281

Definición: Incorrect object accessed: e.g., "X := ABC33" instead of "X := ABD33".

Complemento: Este defecto ocurre debido a un error de concepto del programador. El programador iguala X a ABC33 pensando que esta asignación es correcta, cuando en realidad la variable a asignar era ABD33. Diferenciar esta categoría de la (51xx) Coding and Typographical, en la cual el error anterior se debería a un error de tipeo, en dicho caso el programador quiso asignar ABD33 a X, pero al tipear, escribió C en vez de D.

Ejemplo:

```
File archivo = open("wrong_file.txt");
```

en vez de:

```
File archivo = open("file.txt");
```

4.2.6.2 Access rights violation - 4282

Definición: Access rights are controlled by attributes associated with the caller and the object. For example, some callers can only read the object, others can read and modify, etc. Violations of object access rights are included in this category whether or not a formal access rights mechanism exists: that is, access rights could be specified by programming conventions rather than by software.

4.2.6.3 Data-flow anomaly - 4283

Definición: Data-flow anomalies involve the sequence of accesses to an object: e.g., reading or initializing an object before it has been created, or creating and then not using.

Ejemplo:

```
File f; //intentar leer del archivo sin haberlo abierto
char c = getchar(f);
```

en vez de:

```
File f = open("archivo.txt");
char c = getchar(f);
```

4.2.6.4 Interlock bug - 4284

Definición: Where objects are in simultaneous use by more than one caller, interlocks and synchronization mechanisms may be used to assure that all data are current and changed by only one caller at a time. These are not bugs in the interlock or synchronization mechanism but in the use of that mechanism.

Ejemplo: No se logra la mutuo exclusión de una impresora entre procesos, pero por error de uso del mecanismo de mutuo exclusión, el funcionamiento del mecanismo se supone correcto.

4.2.6.5 Saving or protecting bug - 4285

Definición: Application requires that the object be saved or otherwise protected at different program states, or alternatively, not protected. These are bugs related to the incorrect usage of such protection mechanisms or procedures.

Complemento: Aquí clasifican defectos que se relacionan, por ejemplo, con el mal uso de mecanismos para persistencia de los datos o para controles de acceso. Entre estos defectos se encuentran no persistir los datos que se deben persistir o no proporcionar controles de acceso a determinados objetos que lo necesitan. Observar que el defecto se origina por el mal uso del mecanismo (persistencia o control de acceso), pero el funcionamiento del mismo se considera correcto.

4.2.6.6 Restoration bug - 4286

Definición: Application requires that a previously saved object be restored prior to processing: e.g., POP the stack, restore registers after interrupt. This category includes bugs in the incorrect restoration of data objects and not bugs in the implementation of the restoration mechanism.

4.2.6.7 Access mode, direct/indirect - 4287

Definición: Object is accessed by wrong means: e.g., direct access of an object for which indirect access is required, call by value instead of name or vice versa, indexed instead of sequential or vice versa.

Complemento: No confundir esta categoría con (523x) Data access, en la que el error se manifiesta por no seguir el estándar. El programa puede comportarse en forma correcta aunque no se siga el estándar. Sin embargo, en (4287) el acceso al objeto es erróneo porque no se realiza correctamente, debido al uso de un método de acceso inadecuado.

4.2.6.8 Object boundary or structure - 4288

Definición: Access to object is partly correct, but the object structure and its boundaries are handled incorrectly: e.g., fetching 8 characters of a string instead of 7, mishandling word boundaries, getting too much or too little of an object.

5. IMPLEMENTATION - 5xxx

Definición: Bugs having to do with the implementation of the software. Some of these, such as standards and documentation, may not affect the actual workings of the software. They are included in the bug taxonomy because of their impact on maintenance.

5.1. CODING AND TYPOGRAPHICAL - 51xx

Definición: Bugs which can be clearly attributed to simple coding and typographical bugs. Classification of a bug into this category is subjective. If a programmer believed that the correct variable, say, was "ABCD" instead of "ABCE", then it would be classified as a 4281 bug (wrong object accessed). Conversely, if E was changed to D because of a typewriting bug, then it belongs here.

5.1.1. Coding wild card, typographical - 511x

Definición: All bugs which can be reasonably attributed to typing and other typographical bugs.

5.1.2. Instruction, construct misunderstood - 512x

Definición: All bugs which can be reasonably attributed to a misunderstanding of an instruction's operation or HOL statement's action.

Ejemplo: No saber como se usa el switch.

5.2. STANDARDS VIOLATION - 52xx

Definición: Bugs having to do with violating or misunderstanding the applicable programming standards and conventions. The software is assumed to work properly.

5.2.1. Structure violations - 521x

Definición: Violations concerning control-flow structure, organization of the software, etc.

5.2.1.1 Control flow - 5212

Definición: Violations of control-flow structure conventions: e.g., excessive IF THEN ELSE nesting, not using CASE statements where required, not following dictated processing order, jumping into or out of loops, jumping into or out of decisions.

5.2.1.2 Complexity - 5214

Definición: Violation of maximum (usually) or minimum (rare) complexity guidelines as measured by some specified complexity metric: e.g., too many lines of code in module, cyclomatic complexity greater than 200, excessive Halstead length, too many tokens.

Complemento: Complejidad a nivel de un modulo o función.

5.2.1.3 Loop nesting depth - 5215

Definición: Excessive loop nesting depth.

Ejemplo: La idea del siguiente ejemplo es representar la forma de un anidamiento excesivo de loops.

```
while(cond1){
    ...
    while(cond2){
        ...
        while(cond3){
            ...
            while(cond4){
                ...
            }
            ...
        }
        ...
    }
    ...
}
```

5.2.1.4 Modularity and partition - 5216

Definición: Modularity and partition rules not followed: e.g., minimum and maximum size, object scope, functionally dictated partitions.

Complemento: Muy similar a la subcategoría (5214) Complexity, se diferencian en que la primera es a nivel de un módulo o función, mientas que la segunda es a nivel del sistema, viendo los módulos en su conjunto. Por ello habla de mínimo y máximo tamaño de los módulos, el scope de los objetos, y la forma en que se realizan las particiones en módulos según las funcionalidades.

Ejemplo: Un ejemplo de defecto que entra en esta categoría es que la distribución de funcionalidades entre los módulos tenga baja cohesión y alto acoplamiento. Este caso se convierte en defecto si uno de los fines de la empresa es que la definición de los módulos posea alta cohesión y bajo acoplamiento. Generalmente estas características son deseables en sistemas orientados a objetos.

5.2.1.5 Call nesting depth - 5217

Definición: Violations of component (e.g., subroutine, subprogram, function) maximum nesting depth, or insufficient depth where dictated.

Ejemplo: Este ejemplo muestra la forma de un anidamiento de subrutinas con profundidad máxima.

```
f1(x){
    ...
    f2(x)
    ...
}

f2(x){
    ...
    f3(x)
    ...
}

f3(x){
    ...
    f4(x)
    ...
```

```
}

f4(x){
    ...
    f5(x)
    ...
}

f5(x){
    ...
}
```

5.2.2. Data definition, declarations - 522x

Definición: The form and/or location of data object declaration is not according to standards.

5.2.3. Data access - 523x

Definición: Violations of conventions governing how data objects of different kinds are to be accessed, wrong kind of object used: e.g., not using field-access macros, direct access instead of indirect, absolute reference instead of symbolic, access via register, etc.

5.2.4. Calling and invoking - 524x

Definición: Bug in the manner in which other processing components are called, invoked, or communicated with: e.g., direct subroutine call that should be indirect, violation of call and return sequence conventions.

Complemento: Esta subcategoría es similar a la anterior, pero en vez de tratar objetos de datos (data objects), refiere a subrutinas, funciones, etc.

5.2.5. Mnemonics, label conventions - 526x

Definición: Violations of the rules by which names are assigned to objects: e.g., program labels, subroutine and program names, data object names, file names.

Ejemplo: Una violación de este estilo es que el estándar diga que todas las funciones y procedimientos empiezan con minúscula, y el programa tenga funciones y procedimientos que empiecen con mayúscula.

5.2.6. Format - 527x

Definición: Violations of conventions governing the overall format and appearance of the source code: indentation rules, pagination, headers, ID block, special markers.

5.2.7. Comments - 528x

Definición: Violations of conventions governing the use, placement, density, and format of comments. The content of comments is covered by 53xx, documentation.

Complemento: A diferencia de la sección que viene, en este subgrupo entran aquellos comentarios que no se apeguen al estándar. En el grupo que sigue, los bugs que se encuentran no surgen de la comparación con estándares.

5.3. DOCUMENTATION - 53xx

Definición: Bugs in the documentation associated with the code or the content of comments contained in the code.

5.3.1. Incorrect - 531x

Definición: Documentation statement is wrong.

Ejemplo: Al documentar un algoritmo de recorrida de grafos, se escribe que el algoritmo corresponde al de Dijkstra, cuando en realidad pertenece a Kruskal. Este es un caso en el que la documentación es errónea.

5.3.2. Inconsistent - 532x

Definición: Documentation statement is inconsistent with itself or with other statements.

5.3.3. Incomprehensible - 533x

Definición: Documentation cannot be understood by a qualified reader.

5.3.4. Incomplete - 534x

Definición: Documentation correct but important facts are missing.

5.3.5. Missing - 535x

Definición: Major parts of documentation are missing.

Complemento: La diferencia entre esta categoría y la anterior es que en la primera faltan partes de la documentación, como no escribir lo que hace cada una de las funciones, pero podemos suponer que están documentadas las funciones principales y el cometido de cada módulo. Esta segunda categoría aplicaría al caso en que falta documentar un código entero o una función de importancia.

6. INTEGRATION - 6xxx

Definición: Bugs having to do with the integration of, and interfaces between, components. The components themselves are assumed to be correct.

6.1. INTERNAL INTERFACES - 61xx

Definición: Bugs related to the interfaces between communicating components with the program under test. The components are assumed to have passed their component level tests. In this context, direct or indirect transfer of data or control information via a memory object such as tables, dynamically allocated resources, or files, constitute an internal interface.

6.1.1. Component invocation - 611x

Definición: Bugs having to do with how software components are invoked. In this sense, a “component” can be a subroutine, function, macro, program, program segment, or any other sensible processing component. Note the use of “invoke” rather than “call”, because there may be no actual call as such: e.g., a task order placed on a processing queue is an invocation in our sense, though (typically) not a call.

6.1.1.1 No such component - 6111

Definición: Invoked component does not exist.

Ejemplo: Un módulo llama una función de un módulo A, y el mismo no esta conectado.

6.1.1.2 Wrong component - 6112

Definición: Incorrect component invoked.

Ejemplo: Se llama a la función g del componente A, y en realidad pertenece al componente B.

6.1.2. Interface parameter, invocation - 612x

Definición: Having to do with the parameters of the invocation, their number, order, type, location, values, etc.

6.1.2.1 Wrong parameter - 6121

Definición: Parameters of the invocation are incorrectly specified.

Complemento: El componente llamador pasa parámetros equivocados en la invocación. Los parámetros son erróneos conceptualmente.

Ejemplo: El componente llamado espera recibir el identificador del llamador como parámetro, y el componente llamador le pasa otra característica, distinta de su identificador.

6.1.2.2 Parameter type - 6122

Definición: Incorrect invocation parameter type used.

Complemento: El parámetro que se pasa en la invocación es conceptualmente correcto, pero su tipo es incorrecto.

Ejemplo: El componente llamador debe pasar como parámetro un color. Este componente representa los colores con números hexadecimales. El componente llamado espera recibir un color, pero de tipo string. El defecto esta en el manejo del tipo de los colores en el componente llamador.

6.1.2.3 Parameter structure - 6124

Definición: Structural details of parameter used are incorrect: e.g., size, number of fields, subtypes.

Ejemplo: Pasar al componente llamador un array de 10, cuando espera uno de 100.

6.1.2.4 Parameter value - 6125

Definición: Value (numerical, boolean, string) of the parameter is wrong.

Ejemplo: El componente llamador debe indicar al llamado si ha ocurrido determinado evento, para ello debe pasar como parámetro una variable booleana. Un error en esta categoría es que la invocación se haya programado de forma inversa: cuando ocurre el evento el parámetro que se pasa es falso, y en el caso contrario es verdadero.

6.1.2.5 Parameter sequence - 6126

Definición: Parameters of the invocation sequence in the wrong order, too many parameters, too few parameters.

6.1.3. Component invocation return - 613x

Definición: Having to do with the interpretation of parameters provided by the invoked component on return to the invoking component or on release of control to some other component. In this context, a record, a subroutine return sequence, or a file can qualify for this category of bug. Note that the bugs included here are not bugs in the component that created the return data but in the receiving component's subsequent manipulation and interpretation of that data.

Complemento: Lo retornado por el componente llamado es correcto, pero la manipulación de ese resultado por el componente llamador es incorrecta.

6.1.3.1 Parameter identity - 6131

Definición: Wrong return parameter accessed.

Ejemplo: El componente llamador espera el parámetro de retorno en la dirección de memoria X, y el componente llamado guarda el valor de retorno en la dirección de memoria Y. Se supone X <>Y.

6.1.3.2 Parameter type - 6132

Definición: Wrong return parameter type used. That is, the component using the return data interprets a return parameter incorrectly as to type.

Complemento: Un componente invoca una función de otro esperando en respuesta un resultado cuyo tipo no coincide con el que espera. Cabe destacar que el defecto esta en el componente llamador, que especifica mal el tipo del parámetro que espera, no en el tipo retornado por el componente llamado.

6.1.3.3 Parameter structure - 6134

Definición: Return parameter structure misinterpreted.

Ejemplo: El llamador espera un record que cree tiene determinados campos, cuando en realidad tiene otros.

6.1.3.4 Return sequence - 6136

Definición: Sequence assumed for return parameters is incorrect.

Ejemplo: El invocador espera recibir los parámetros en un orden que no es, produciéndose un error después en el uso de los mismos.

6.1.4. Initialization, state - 614x

Definición: Invoked component not initialized or initialized to the wrong state or with incorrect data.

Ejemplo: Invocación a un módulo para que contabilice determinados eventos, si no se *resetea* el contador a cero, va a ser erróneo.

6.1.5. Invocation in wrong place - 615x

Definición: The place or state in the invoking component at which the invoked component was invoked is wrong.

Complemento: Un componente invoca a otro en momento equivocado.

Ejemplo: Los parámetros que se pasan en la invocación no han recibido el procesamiento necesario para que la invocación tenga sentido.

6.1.6. Duplicate or spurious invocation - 616x

Definición: Component should not have been invoked or has been invoked more often than necessary.

Ejemplo: Un sistema cuenta con un componente que realiza impresiones de la información que recibe del resto de los componentes del sistema. Un defecto seria que componentes que realizan cálculos auxiliares para los componentes principales llamen al componente de impresión, el error esta en que los módulos auxiliares no cuentan con todos los datos relacionados con la composición de una impresión. Este caso sería invocar a un componente cuando este no debía serlo. El caso de llamar un componente mas frecuentemente que lo necesario seria que los componentes principales llamen al de impresión diariamente, cuando debería ser semanalmente, generando impresiones inútiles a los fines del usuario.

7. Conclusiones

Aquí se presentan los tipos de defectos que quedaron sin comprenderse completamente y las conclusiones generales acerca de la Taxonomía de Beizer.

7.1. Tipos de Defectos Incompletos en el Análisis

El grupo (3xxx) Structural Bugs, parece estar referido a los defectos en la estructura del código pero, según Beizer, el código es sólo un ejemplo de aplicación de este grupo. Se podría pensar que ciertos defectos en el diseño podrían ser de esta categoría, sin embargo, entendemos que cualquier defecto en el diseño corresponde indefectiblemente a la categoría (2xxx) Fuctionality as Implemented. Defectos en la arquitectura de software pertenecen al grupo (7xxx) System and Software Architecture. Entonces, no queda claro qué otros defectos estructurales pueden existir sin ser en el código.

En la Taxonomía no se diferencia en forma clara un caso “don’t care” (3124) de un caso “illogical” (3126). Beizer plantea que un caso “illogical” es mas fuerte que un “don’t care”. Un defecto del tipo “illogical” es aquel en que se ha asumido incorrectamente que algo no puede ocurrir.

La definición de caso “don’t care” incluye casos imposibles, por lo que la diferencia entre ambos casos queda poco clara.

A pesar de que se ha agregado un complemento a la categoría Code conversion (3226.6), para que no se confunda con Type transformation (4214), no queda claro a que tipos de defectos se refiere Beizer con “code conversion”. Diferenciando la categoría de la (4214), podrían ser defectos asociados a la conversion de código ASCII a Unicode, por ejemplo. Pero no tenemos la certeza de que Beizer se refiera a este tipo de conversion de código.

La definición de la categoría (3228) Symbolic, algebraic, hace difícil entender a que tipo de defectos quiso clasificar en ella Beizer cuando la agrego a su taxonomía. “Bugs in symbolic processing of algebraic expressions”.

En la categoría (611x) Component invocation, no se ha logrado diferenciar los conceptos de *invoke* y *call*.

7.2. Conclusiones Generales

El tamaño de la taxonomía, así como su nivel de granularidad y numerado no uniforme de las categorías, hacen que sea difícil de aplicar y confusa a la hora de utilizarla. Durante el análisis, al buscar ejemplos para las categorías poco claras muchas veces surgió la interrogante de si dicho ejemplo realmente aplicaba a la categoría para la cual había sido pensado y no en otra, o si no sería mas apropiado que clasificara en las dos. Como se parte de la base que las categorías y subcategorías son excluyentes entre si surge la dificultad de decidir en cuál clasificar determinado defecto, para algunos casos.

Sin embargo, el nivel de granularidad de los tipos de defectos es interesante para nuestra investigación. Esta granularidad permitiría conocer si algún tipo de defecto muy particular no es detectado con alguna técnica de verificación pero sí con otras.

Entonces, entendemos que esta clasificación debe ser usada en nuestros experimentos pero no debe ser la única en utilizarse. Otras taxonomías mas sencillas pueden agilizar y facilitar nuestra investigación y deben también ser consideradas.

Referencias

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990. 1, 2

[2] IBM. Orthogonal defect classification. web page http://researchweb.watson.ibm.com/softeng/ODC/ODC.HTM. 1