Universidad de Talca
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Construcción de Software

Prueba práctica II

Erik Regla
eregla09@alumnos.utalca.cl

29 de Junio del 2017

1. Pregunta 5 - Verificación y validación

1.1. Indicar qué patrones de diseño (codificación) podrían ser usados para desarrollar el sistema. Justifique su respuesta

Considerando los patrones de diseño vistos en clases (limitando la lista a los presentados), es posible clasificar los diferentes patrones en "patrones" y "antipatrones".

Patrones de diseño aplicables:

- **Strategy:** Se tiene conocimiento de las implementaciones para el cálculo. En este caso en particular las implementaciones pueden no variar mucho, sin embargo en el tiempo de ser reemplazadas o de incluirse nuevos criterios para el cálculo de la tarifa, Strategy puede suponer una ventaja.
- **Proxy:** No existe suficiente información para afirmar que el cálculo de la tarifa se realizará dentro del mismo módulo (dado que es solo alquiler), si existe un módulo separado el cual se encarga de tarifar, entonces el uso de proxy supone una ventaja a la hora de integrar estos.

Antipatrones de diseño observados:

- **Patrones creacionales como Abstract Factory, Factory Method, Adapter , Singleton y Composite:** Si bien la portabilidad de elementos de diseño es una característica deseable, el problema se centra en el cálculo de la tarifa, por lo que la aplicación de este tipo de patrones solo incrementaría la complejidad en la mayoría de los casos al ser un conjunto de patrones creacionales.
- **Template:** Se tienden a utilizar en conjunto con patrones creacionales del tipo Factory, sin embargo dado que estas no aplican (por el momento, no hay razón para implementar un sistema de plantillas.
- **Observer:** No hay interdependencia de eventos visible dentro de la función.

- **Null Object:** No es aplicable.

1.2. Realizar una tabla con las clases de equivalencia indicando las clases válidas y no válidas para cada variable de entrada. Enumerar las clases obtenidas.

Cuadro 1: Tabla de clases de equivalencia

Variable	Clases de equivalencia válidas	Clases de equivalencia inválidas
Nombre	$\in$ [a-zA-Z]+	$\notin$ [a-zA-Z]+
Password	$\in$ ^(?=.*[a-zA-Z])(?=.*[\d])[a-zA-Z\d]{8,20}	$\notin$ ^(?=.*[a-zA-Z])(?=.*[\d])[a-zA-Z\d]{8,20}
Edad	$\in \mathbb{N}$	$\notin \mathbb{N}$
Tipo_de_socio	$\in (Gold \vert Silver \vert Standard)$	$\notin (Gold \vert Silver \vert Standard)$

1.3. Obtener un conjunto finito de casos de prueba significativos para dicha tabla, indicando qué clases de equivalencia cubriría cada caso.

Cuadro 2: Tabla de casos de prueba

Variable	Entradas válidas	Entradas inválidas
Nombre	{hiss|purr}	{hiss1|ϵ}
Password	$\{hisshisspurr666\}$	$\{hisshisspurr \vert 6666666666 \vert a \vert$ $q1w2e3r4t5y6u7i8o9p0asdf\}$
Edad	$\{1, 2, 20, 99\}$	$\{-1, -2, -20, -99, 0\}$
Tipo_de_socio	$\{Gold, Silver, Standard\}$	$\{Extended, Premium, Purr\}$

2. Pregunta 6 - Verificación y Validación

2.1. Pseudocódigos

Algorithm 1 Comparación de misma ciudad entre dos tuplas

```
1: procedure MISMA_EDAD(U1,U2)
2:     return usuarios[u1][1] = usuarios[u2][1]
```

Algorithm 2 Diferencia de edad entre dos tuplas

```
1: procedure DIFERENCIA_EDAD(U1,U2)
2:     return |usuarios[u1][2][0] − usuarios[u2][2][0]|
```

Algorithm 3 Obtencion de relaciones de amistad para una tupla

```
1: procedure OBTENER_AMIGOS(U)
2:     A ← ∅
3:     for t ∈ amistades do
4:         if t[0] = u then
5:             A ← A ∪ {t[0]}
6:         if t[1] = u then
7:             A ← A ∪ {t[0]}
8:     return A
```

Algorithm 4 Recomendacion de amigos para una tupla

```
1: procedure RECOMENDAR_AMIGOS(U)
2:     A ← obtener_amigos(u)
3:     R ← ∅
4:     for t ∈ A do
5:         for p ∈ obtener_amigos(t) do
6:             if p ∉ A ∧ misma_ciudad(u, p) ∧ diferencia_edad(u, p) < 10 ∧ (u ≠ p) then
7:                 R ← A ∪ {p}
8:     return R
```

2.2. Grafo de flujo

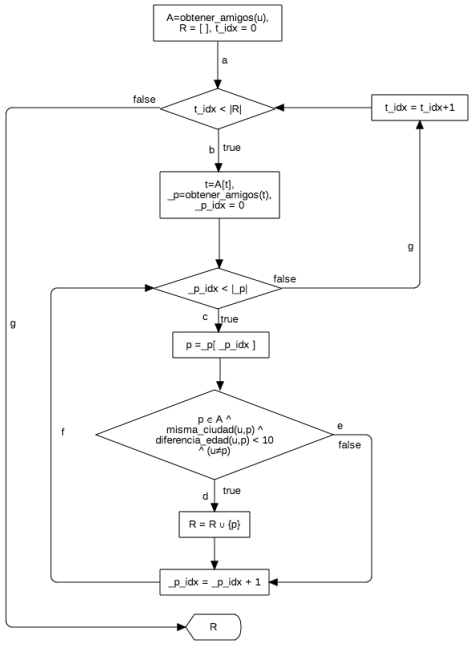

Figura 1: Grafo de flujo para la función *recomendar_amigos*(u)

2.3. Ejecución simbólica

$$
\begin{aligned}
&[x = X, y = Y, a = A, r = 0] \\
&[x = A[X], y = A[Y]] \\
&[x = A[X][2], y = A[Y][2]] \\
&[x = A[X][2][0], y = A[Y][2][0]] \\
&[r = A[X][2][0] - A[Y][2][0]]]
\end{aligned}
\tag{1}
$$

Como es posible apreciar, el código falla si es que tanto $X(u1)$ como $Y(u2)$ no poseen la estructura de tupla presentada en el enunciado.

Figura 2: Ejecución simbólica para $diferencia\_edad(u1, u2)$

2.4. Cobertura de sentencias

```
usuarios = {
    522514: ["Jean_Dupont", "Marseille", [1989, 11, 21]],
    587125: ["Perico_Los_Palotes", "Valparaíso", [1990, 4, 12]],
    189471: ["Jan_Kowalsky", "Krakow", [1994, 4, 22]],
    914210: ["Antonio_Nobel", "Valparaíso", [1983, 7, 1]]
}
amistades = [
    [198471, 289142], [138555, 429900], [349123, 781118], [522514, 587125], [189471, 914210], [587125, 189471]
];
u = 587125
```

Figura 3: Caso de prueba que cubre la secuencia abcecfbcdccecfg para $recomendar\_amigos(u)$

2.5. Cobertura de condición

```
usuarios = {
    522514: ["Jean_Dupont", "Marseille", [1989, 11, 21]],
    587125: ["Perico_Los_Palotes", "Valparaíso", [1990, 4, 12]],
    189471: ["Jan_Kowalsky", "Krakow", [1994, 4, 22]],
    914210: ["Antonio_Nobel", "Valparaíso", [1983, 7, 1]]
}
amistades = [
    [198471, 289142], [138555, 429900], [349123, 781118], [522514, 587125], [189471, 914210], [587125, 189471]
];
u = 587125
```

Figura 4: Caso de prueba que cubre la secuencia abcecfbcdccecfg para $recomendar\_amigos(u)$

2.6. Cobertura de flujo de datos

Variable **u**:

```
1  function obtener_amigos(u) {
2      var a, r;
3      a = amistades;
4      r = new Array();
5      var a_idx = 0;
6      while(a_idx < amistades.length){
7          var t = amistades[a_idx];
8          if(t[0] === u){
9              r.push(t[1]);
10         }
11         if(t[1] === u){
12             r.push(t[0]);
13         }
14     }
15     return r;
16 }
```

Figura 5: Expansión de la implementación de *obtener_amigos*(u)

- **Definiciones de u:** 1
- **C-Uso de u:** 0
- **P-uso de u:** 6,8
- **Caminos Libre definición de u:** (0,1,2,3,4,12), (0,1,2,3,4,5,6,7,8,9,10,11), (0,1,2,3,4,5,6,8,9,10,11), (0,1,2,3,4,5,6,7,8,10,11), ...
- **Camino-du:** (0,1,2,3,4,5,6) establece una definición-p-uso (1,(6,t),u) y (1,(6,f),u)

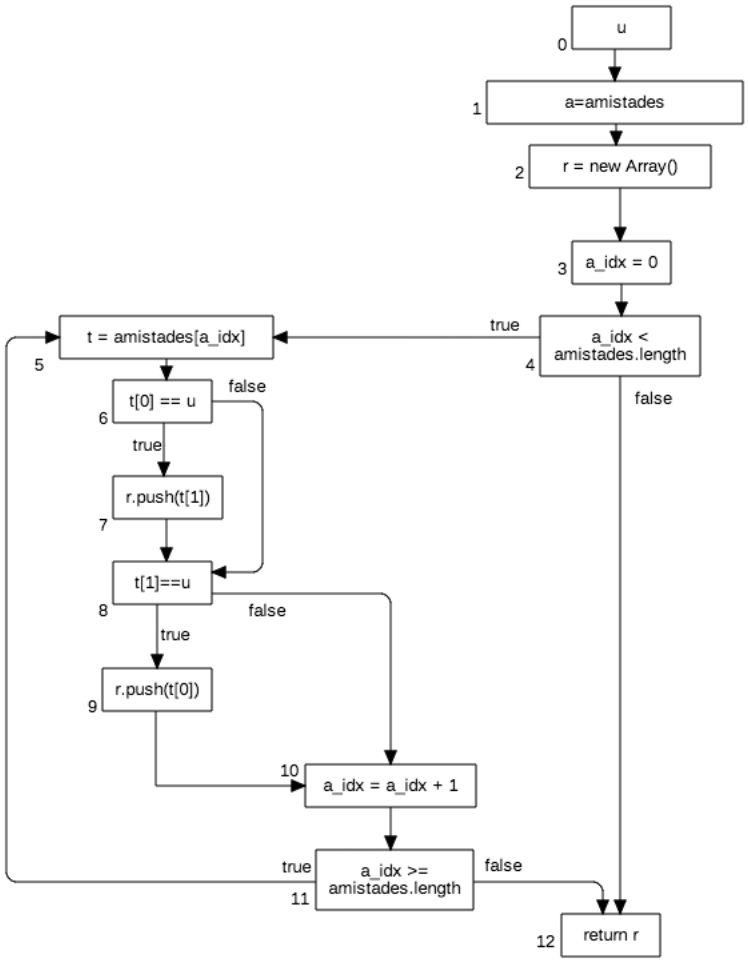

Figura 6: Grafo de flujo para la función *obtener_amigos*(u)

2.7. Bonus - Código implementación

```javascript
'use strict';

var usuarios = {
    522514: ["Jean Dupont", "Marseille", [1989, 11, 21]],
    587125: ["Perico Los Palotes", "Valparaíso", [1990, 4, 12]],
    189471: ["Jan Kowalsky", "Krakow", [1994, 4, 22]],
    914210: ["Antonio Nobel", "Valparaíso", [1983, 7, 1]]
}

function misma_ciudad(u1, u2) {
    return usuarios[u1][1] === usuarios[u2][1];
}

function diferencia_edad(u1, u2) {
    return Math.abs(usuarios[u1][2][0] - usuarios[u2][2][0]);
}

var amistades = [
    [198471, 289142], [138555, 429900], [349123, 781118]
    // Extra friends for testing
    , [522514, 587125], [189471, 914210], [587125, 189471]
];

function obtener_amigos(u) {
    let a = amistades.map(t => {
        if (t[0] === u) return t[1];
        if (t[1] === u) return t[0];
    }).filter(t => t);

    return (a)? a : new Array();
}

function recomendar_amigo(u) {
    let a = obtener_amigos(u);
    let r = new Array();
    a.map(t => {
        obtener_amigos(t).map(p => {
            if (
                (!(a.indexOf(p) >= 0)) && //is not friend - prevent self looping
                (misma_ciudad(u,p)) && //same city
                (diferencia_edad(u,p) < 10) && //different age
                (u!==p) //assumed. Since we can target onself
            ) r.push(p);
        });
    });
    return r;
}

//Unit testing?
if (misma_ciudad(587125, 522514) === false && misma_ciudad(587125, 914210) === true)
        console.log("misma_ciudad_OK");
else console.log("misma_ciudad_FAIL");

if (misma_ciudad(522514, 587125) === false && misma_ciudad(914210, 587125) === true)
    console.log("misma_ciudad_(reversed)_OK");
else console.log("misma_ciudad_(reversed)_FAIL");

if (diferencia_edad(914210, 587125) === 7 && diferencia_edad(522514, 587125) === 1)
    console.log("diferencia_edad_OK");
else console.log("diferencia_edad_FAIL");

if (diferencia_edad(587125, 914210) === 7 && diferencia_edad(587125, 522514) === 1)
    console.log("diferencia_edad_(reversed)_OK");
else console.log("diferencia_edad_(reversed)_FAIL");

if (obtener_amigos(522514).sort().join(',')===[587125].sort().join(','))
    console.log("obtener_amigos_OK");
else console.log("obtener_amigos_FAIL");

if (recomendar_amigo(587125).sort().join(',')===[914210].sort().join(','))
    console.log("recomendar_amigo_OK");
else console.log("recomendar_amigo_FAIL");
```

Figura 7: Código Javascript que implementan las funciones 1, 2, 3, 4 y 5