الإتقان في البرمجة

ابن العيد

مقدمة

يتصور العديد من المطورين أن مراحل تطوير البرمجيات تنتهي بمجرد إنشاء ملفات التنصيب متجاهلين بذلك عدة مسائل قد تكون أكثر أهمية في مراحل التطوير. هذه الأسطر كانت نتيجة لإنتقالي إلى التطوير في نظام التشغيل Linux بلغة الـ C بعد فترة من التطوير في بيئة الـ DotNet في نظام التشغيل windows. قبل الشروع في صلب الموضوع أذكر المطورين بمفهوم الإتقان, هذا المفهوم الذي يمتد على ثلاثة مراحل هامة تجعل من المطور محترفا و تطيل في مراحل التطوير لتنتج في نهاية المطاف برنامجا متقنا و تنافسيا. ككخطوة أولى يجب توفير الوضائف الرئيسية للبرنامج و التي تم إبرازها بالخط العريض في دفتر الشروط, المرحلة الموالية تتمثل في جعل البرنامج يعمل بشكل جيد و بدون ثغرات و أخيرا ينبغي الحرص على سرعة عمل البرنامج المطور.

1. إجعل البرنامج يعمل

هذا المقال لا يهدف لتعليم البرمجة أو كيفية تطوير البرمجيات إنما يحاول تسليط الضوء على مسائل أكثر أهمية تخص بالأساس جودة البرنامج المطور و سرعته. لذلك أذكر بضرورة الإكتفاء في هذه المرحلة بتوفير المطلوب من مهام دون البحث عن الجمالية، تنسيق الشفرة و طولها و غير ذلك مما يمكن أن يشغل المطور. هذه المرحلة هي الأطول و الأصعب في عمر تطوير البرمجيات و ترتبط أساسا بخبرة المطور و مدى إتقانه لتصميم البرنامج[1] و متطلباته الأولية و حرصه على وضوح الشفرة التي يكتبها. لذلك، من العادات الحسنة التي يحبذ تبنيها عند كتابة الشفرة إضافة التعليقات الضرورية التي تفسر التعليمات الصعبة، سبب إسناد قيمة معينة لأحد المتغيرات إلى غير ذلك مما ترى ضرورة شرحه. إضافة التعليقات لا يقتصر على التعليمات فقط ، إنما من الضروري إدراجها في بداية كل ملف لذكر وضيفة محتوياته، المشروع الذي ينتمي له، كاتبه وغير ذلك كما هو مجسد في الصورة رقم 1.

[1] تصميم البرنامج :Software modeling.

```
/** ****************************************************************************
 *  Copyright: ……                                                             *
 *  Project: Ptf Control and Algorithms Rel.1                                  *
 *  File: ChannelSPF_EFT.c                                                     *
 *  $Revision: 1.6 $                                                           *
 *  Date: 08/07/15                                                             *
 *  SDD component: External Interface Module                                   *
 *  Purpose: Implementation of EFT protocol.                                   *
 *                                                                             *
 *  Implemented Requirements: N/A                                              *
 *  Language: English                                                          *
 *  Author: HD                                                                 *
 *  History:                                                                   *
 *-----------------------------------------------------------------------------*
 * $Log: ChannelSPF_EFT.c,v $
 * Revision 1.6  2008/09/10 13:27:43  ……
 * Substitute G_Connection netbuf structutre by the external G_FtpConnectionHandler.
 *……
 * ****************************************************************************/
```

الصورة رقم 1: التقديم للملف المصدري.

نفس الشيء بالنسبة للدوال، لابد أولا من تسميتها بأسماء متطابقة مع وظائفها مع تعليق كاف يجعل غيرك غير مجبرعلى تتبع تعليمات الدالة لفهم وظيفتها.

```
/**
 * Name:        ConstructAndValidate_EFT_UploadData_Signal().
 * Purpose:     Construct EFT_UploadData signal and validate its XSD.
 * Argument I/O: I: FullPath local repository
 *              O: LOG_ERROR variable:SUCCESS if both construction and validation
 *               were performed successfully.
 * Author:      HD
 */
 LOG_ERROR ConstructAndValidate_EFT_UploadData_Signal(char *FullPath)
{   LOG_ERROR FuncReturn=FTP_SOCKET_ERROR;
//. .
}
```

الصورة رقم 2: التقديم للدالة.

هذه القواعد لا تخص فقط المبرمجين بلغتي الـــ C و الـــ C++ فقط ، حتى مبرمجي الـــ DotNet تخصهم هذه القواعد حيث يتمتع المبرمجون بالـــ DotNet بالكثير من الرفاهية كتعدد أقسام الاستثناءات مثل DivideByZeroException, DllNotFoundException, IOException و ململم النفايات[2] و العديد من الإمتيازات الأخرى.

يحبذ في منصة الـــ DotNet تخصيص أقسام مستقلة لتعريف الدوال الأصلية[3] و قسم خاص لكل لائحة أو تعديد[4]، بهذه الاستقلالية يتيسر على المبرمج الوصول لهذه اللائحة. في حين أن دمجها في شفرة قسم أو قسام أخرى يجبر المطور على تعريفها بصفة pubic و التمهيد للقسم أو الأقسام المنتمية لها، مما يؤدي لتنفيذ تعليمات زائدة و ربما غير مرغوب فيها وقتها.

[2] ململم النفايات: Garbage Collector.
[3] الدوال الأصلية: Native APIs.
[4] تعديد: Enumeratio Type.

2. إجعل البرنامج يعمل جيدا

بعد توفير الجزء الرئيسي من البرنامج و التأكد من إنجاز الوضائف الرئيسية المطلوبة, يمكن الإنتقال للمرحلة الموالية والمتمثلة في جعل البرنامج يعمل جيدا/ مع التأكيد على كلمة جيدا. من أبرز المهام المطلوبة من المطور في هذه المرحلة معالجة الاستثناآت. فينبغي على المطور تصور كل السيناريوهات غير المتوقعة ؛ كغياب إحدى الموارد التي يعتمدها البرنامج من ملفات و مكتبات و غير ذلك. نضرب مثالا آخر و المتمثل في مراعات نظم التشغيل التي قد يستخدم فيها البرنامج و مدى مطابقتها للنظام المعتمد في التطوير ، تتعدد الأمثلة و تتنوع في هذا المجال.

جودة البرنامج مرتبطة إرتباطا وثيقا بجودة كتابة الشفرة و مدى وضوحها. هذه النقاط تفرض نفسها عند المساهمة في مشروع ضخم و في إطار فريق عمل متكامل, حيث تصبح كل شفرة كتبتها أو كتبها أحد زملائك مشتركة و قابلة للتطوير و للنقد اللاذع أيضا. لذلك تم توفير عدة برامج لمشاركة الشفرة مع زملائك و مراقبة مراحل التطوير, من أبرز هذه البرمجيات; CVS SourceSafe,و البرنامج الباهض جدا ClearCase, مثل هذه البرمجيات تجبرك على الحرص -كل الحرص- على جودة الشفرة التي تكتبها وكل إصدار تشارك به[5]، خاصة إذا وجدت نفسك محاصرا بقائمة من القواعد لكتابة الشفرة بكيفية متفق عليها.

غالبا ما تعد بعض هذه القواعد بديهية و لا داعي حتى لذكرها و لكن ينبغي التذكير أن هذه القواعد تطرح بشكل عام لمراعات الاختلاف بين المترجمات و حتى الإختلاف بين إصداراتها. فلو جربت مثلا ترجمة نفس الشفرة في الفيجوال ستوديو وبالمترجم gcc, ستجد أن هناك تحذيرات يبرزها مترجم و يتجاهلها الآخر و بطبيعة الحال في هذا المثال ينبغي أن تكون الشفرة مستقلة على نظامي التشغيل و مكتباتهما الخاصة. الحديث هنا عن التحذيرات و ليس الأخطاء، لأن تصويب الأخطاء يتم تجاوزه بمجرد الانتهاء من المرحلة السابقة في التطوير و نبحث الآن عن جودة الشفرة.

هذا جزء صغير من لائحة قواعد كتابة الشفرة التي و ضعت أمامي عند إلتحاقي بفريق العمل الحالي:

Rule C4 (DAL C,D)- - Names representing user defined types shall be in mixed case starting with upper case and ending with suffix '_T':
Line_T, BusMessage_T
Rule C5 (DAL C,D)- - Variable names must be in mixed case starting with upper case.
Line , BusMessage
Rule C6 (DAL C,D)- - Types and variables shall not start with an underscore character.
Rule C7 (DAL C,D)- - Named constants (including enumeration values) must be all uppercase using underscore to separate words.
MAX_ITERATIONS, COLOR_RED, PI
Rule C8 (DAL C,D)- - Pointers variables shall be prefixed by 'Pt'
BYTE *PtTemperature

الصورة رقم 3: بعض قواعد كتابة الشفرة.

بما أن العالم ليس مثاليا و جدت نفسي مطالبا بملاحقة إنتهاكات زملائي لهذه القواعد و تصويبها (و هي مهمة شاقة أمام شفرة تتكون من آلاف الأسطر), و للخروج من هذه "الورطة" إستخدمت الأدات Splint [1] التي تستخدم لتتبع الأخطاء اللغوية في الشفرة خارج وقت الترجمة. لا أعني بالأخطاء اللغوية ذلك المفهوم المجرد المعتمد في اللغات الحية إنما الأخطاء والتجاوزات التي قد يكتشفها ال pre-compiler . تسهل هذه الأداة تتبع الأخطاء مع إمكانية تعديلها و تطويعها و للأسف فإن تعديلها معقد

[5] إصدار تشارك به :Checked in version.

و لا يوجد معلومات كافية عن ذلك على شبكة الأنترنت مما دفعني لتطوير سكريبت Perl يقوم بتتبع التجاوزات التي تجاهلها الـ Splint.

تجسد الصورة رقم 4 مثالا على مخرجات الأدات بعد استخدام ما يفوق الخمس عشر مدخلا[6] لتعديل عمل الأداة كتجاهل بعض أنواع الأخطاء و تبين لغة البرمجة المستخدمة (Ansi C) و إجبار الأدات على محاولة مواصلة البحث عن الأخطاء عند تعطلها

```
../../ptf_v1/src/modules/ExtInt/ExtInt.c:286:26:
  Storage G_ClientListOfUP may become null
../../ptf_v1/src/modules/ExtInt/ExtInt.c: (in function SubTimer)
../../ptf_v1/src/modules/ExtInt/ExtInt.c:366:48:
   Function pthread_create expects arg 3 to be [function (void *) returns void
   *] * gets void *: G_PtSubThreadFunction
 Types are incompatible. (Use -type to inhibit warning)
../../ptf_v1/src/modules/ExtInt/ExtInt.c: (in function main)
../../ptf_v1/src/modules/ExtInt/ExtInt.c:827:28:
   Assignment of [function (void) returns void *] to void *:
   G_PtSubThreadFunction = ThDwlExecutor
../../ptf_v1/src/modules/ExtInt/ExtInt.c:832:60:
```

الصورة رقم 4: مثال لمخرجات البرنامج Splint

أحيانا تفشل الأداة Splint على العمل بشكل اعتيادي لعدم تحديد مسار المكتبات التي تعتمدها مرجعة نصا محددا يتمثل في:

Check LARCH_PATH environment variable

وهو ما يوجب تعديلها عبر تنفيذ الأمرين التاليين:

```
LARCH_PATH=~/splint-3.1.1/lib:/usr/lib:/usr/local/lib
LCLIMPORTDIR=~/splint-3.1.1/imports
```

ملاحظة: الغاية من الملاحظات السابقة تسهيل الأمر على من يريد تجربة الأدات و حتى تطويرها (الأداة مفتوحة المصدر.)

نعود للمشكلة الأكثر تعقيدا و المتمثلة في تتبع مساحات الذاكرة المحجوزة و غير المفرغة, يمكن تتبع هاته الأخيرة باستخدام الأدات Valgrind [2] التي تبرز المتغيرات التي يجب إفراغ قيمتها. لكن بهذا النحو تم حل جزء بسيط من الإشكال و لا يزال الجزء الأوفر حيث ينبغي البحث عن المكان الأمثل لإضافة تعليمة الإفراغ, لأن أغلب المتغيرات مشتركة بين عدة أقسام من المشروع إضافة إلى الاستخدام المتكرر لهذه المتغيرات مم قد يسبب مشاكل من نوع استخدام مؤشرات غير محجوزة أو لا تحتوي قيما أولية. الإشكال الثاني يتمثل في مدى صحة عملية الإفراغ:

[6] مدخل: Flags.

```
// INCLUDES
#include <stdlib.h>
#include <stdio.h>
// TYPES
typedef struct FileInfo
 {char FullPath[50];
  char FileName[20];
  char AccessPermission[10];
  long long int size;
 };
// CONSTANTS
#define READ "F_CANREAD"
// METHODS
int main()
{   FileInfo *EFT_Upload;
    char* FileTempExtension;
    // Memory allocation
    FileTempExtension=(char*)malloc(sizeof(FileTempExtension));
    EFT_Upload =(FileInfo*)malloc(sizeof(struct FileInfo));

    // populate the EFT_Upload structure
    sprintf(EFT_Upload->FileName,"EFT_Upload.xml");
    sprintf(EFT_Upload->FullPath,"/home/MyName/Boxes/EFT_Upload.xml");
    sprintf(EFT_Upload->FullPath,READ);
    // Assign a value to the char* variable FileTempExtension
    strcpy(FileTempExtension,".temp");

    // Continue your work here …

    free(FileTempExtension);
    free(EFT_Upload);  // Invalid structure free operation !!!
    return 0;
}
```

الصورة رقم 5: إفراغ قيم المؤشرات المحجوزة.

تجسد الصورة رقم 5 شفرة مثال قمت فيه بحجز متغيرين في ذاكرة الحاسوب. المتغير الأول من نوع char * في حين أن المتغير الثاني عبارة عن لائحة (Structure) بمجموعة من المتغيرات الفرعية لا تحتوي مؤشرات. عندما تتكون اللائحة من مجموعة من المؤشرات فإن إفراغ الواحد تلوى الآخر ينتهي بإفراغ اللائحة, و لكن إذا كانت المتغيرات من نوع غير المؤشر كمصفوفة من الحروف مثلا ، فإن عملية الإفراغ لا تكون سليمة و استخدام الدالة delete لا يكون في محله عكس المثال التالي:

```
// INCLUDES
#include <stdlib.h>
#include <stdio.h>
// TYPES
typedef struct FileInfo
 {char * FullPath;
  char * FileName;
  char * AccessPermission;
  long long int size;
 };
// CONSTANTS
#define READ "F_CANREAD"
// METHODS
int main()
{   FileInfo *EFT_Upload;
    char* FileTempExtension;
    // Memory allocation
    FileTempExtension=(char*)malloc(sizeof(FileTempExtension));
    EFT_Upload =(FileInfo*)malloc(sizeof(struct FileInfo));

    // populate the EFT_Upload structure
    strcpy(EFT_Upload->FileName,"EFT_Upload.xml");
    strcpy(EFT_Upload->FullPath,"/home/MyName/Boxes/EFT_Upload.xml");
    strcpy(EFT_Upload->FullPath,READ);
    // Assign a value to the char* variable FileTempExtension
    strcpy(FileTempExtension,".temp");

    // Continue your work here …

    free(FileTempExtension);
    free(EFT_Upload->FileName);
    free(EFT_Upload->FullPath);
    free(EFT_Upload->FullPath);
    // Now EFT_Upload structure is free
    return 0;
}
```

الصورة رقم 6: إفراغ قيم المؤشرات المحجوزة.

يبرز المثالان السابقان بعض الإشكاليات التي يمكن أن يواجهها المطور عند سعيه لإفراغ قيم المؤشرات المحجوزة و المفروغ منها, حيث أن الإشكال الرئيسي كما ذكرت لا يكمن في معرفة المؤشرات التي لم يتم إفراغ قيمتها بل في التعامل معها بشكل صحيح و وضع تعليمات الإفراغ في المكان الصحيح.

3. إجعل البرنامج يعمل بسرعة

هذا العامل يبرز و بشكل جلي في البرامج الضخمة و المتعلقة بمعالجة الصور و الشبكات. حل هذا الإشكال يسير إلى حد ما ويتمثل أساسا في تقليص المساحات المحجوزة من الذاكرة إلى أقصى حد ممكن و التي لم يتم إفراغها ما إن يتم الفراغ منها. هذا الإشكال تم حله في بيئة الـــ DotNetعبر توفير ململم النفايات و لكن معالجة هذه المسألة عند البرمجة بالـــ C أو الـــ C++ يصعب, حيث يجبر المطور على تتبع مساحات الذاكرة المحجوزة حتى نهاية وضيفة البرنامج و التي يمكن أن تتراكم حتى تشل عمله.

مثال آخر في كيفية التسريع من عمل البرنامج يكمن في التخلص من الأجزاء عديمة الفائدة من الشفرة إن وجدت، بالإضافة إلى الاستغلال المتكرر للملفات و المكتبات و إجتناب الإستعمال المكثف للحلقات. غالبا ما لا ننتبه لهذا الأخير, فلو فرضنا أن المطور بصدد القراءة من ملف و البحث عن جملة معينة, فبمجرد عثوره عليها يؤشر لذلك دون إيقاف الحلقة التي ستكون مواصلة بحثها عقيما، و في الآن ذاته مضيعة كبيرة للوقت. المطور هنا لا يتعامل مع متغيرات و بيانات محلية تابعة لشفرته إنما خارجية مما يعني تراكما في الذاكرة المحجوزة و مزيدا من الوقت الضائع و الموارد المهدورة. الوقت الضائع بسيط جدا و لكن في مجال الحاسب, الجزء المئوي من الثانية كاف للقيام بالعديد و العديد من المهام. قد تسند لك في إطار عملك مهمة البحث في أسباب بطئ برنامج مهدد بالنقل للأرشيف، كما كان حال ستيف أولين[3][4] حيث كان لإختيار نوع المتغيرات والدوال ، استخدام الدالة scanf و نوع الملفات التي يأخذ منها البرنامج البيانات أثرا كبيرا في بطئ البرنامج.

خاتمة

أختم هذا المقال بالتذكير أن إتقان صناعة البرمجيات يمر بعدة مراحل يكتشفها المبرمج فقط عند المشاركة في مشاريع ضخمة، حيث تكون الأدوات اللازمة مثل CaseClear و Electric Coud متوفرة. لكن يبقى المطور المتقن لفنون كتابة البرمجيات العقل المدبر و النواة الرئيسية التي قد تنجح المشروع بعد أن كان مهددا بالفشل.

المراجع

[1] www.splint.org
[2] www.valgrind.org [2]
[3] Practical C++ Programming, Steve Oualline, Page 316
[4] http://www.arabteam2000-forum.com/index.php?showtopic=183302&st=0#entry922141

تم و لله الحمد...