基于COCOMOⅡ的自动测试维护代价实例研究

阚红星[1,2],马溪骏[1],桂宏新[1]
(1.合肥工业大学 管理学院,安徽 合肥 230009;
2.合肥学院 计算机系,安徽 合肥 230022)

摘 要:不考虑自动测试的维护代价,盲目进行自动化测试是有风险的。文中以画板程序升级为例,详细介绍了它的修改过程和相应测试程序的维护代价,利用COCOMOⅡ度量方法,对测试程序和被测试程序的维护代价做一个比较,说明测试程序和被测试程序一样,需要反复维护才能进行回归测试。通过实例和理论分析使测试者清醒认识自动测试的维护代价,合理利用自动化测试。

关键词:软件维护;回归自动测试;维护代价;COCOMOⅡ
中图分类号:TP311.56 **文献标识码**:A **文章编号**:1673-629X(2008)11-0047-04

A Case Study on Maintenance Cost for Regression Test Automation Based on COCOMOⅡ

KAN Hong-xing[1,2], MA Xi-jun[1], GUI Hong-xin[1]
(1. School of Management, Hefei University of Technology, Hefei 230009, China;
2. Department of Computer, Hefei University, Hefei 230022, China)

Abstract: Carrying on the test automation blindly will have the risk, without considering the test automation maintenance cost. Taking the drawing board as the example, its revision process and the corresponding test program maintenance cost was introduced in detail. Comparing the test program maintenance cost with its test object maintenance by COCOMOⅡ measurement method, automation regression test must be maintained repeatedly like its test object before testing. Test automation maintenance cost will be known soberly and test automation will be used reasonably by example and the theoretical analysis.

Key words: software maintenance; regression test automation; maintenance cost; COCOMOⅡ

0 引 言

随着软件业的迅速发展,软件测试技术的自动化是软件测试的发展趋势。在某些情况下,自动测试确实有自己优势,如在分布式系统测试[1]、完整的代码覆盖测试[1]、大容量数据测试[2]中,手工测试很难做到完美,甚至根本无法测试。但不能因此而认为自动测试就是解决测试问题的"银弹"(Silver Bullet)[3]。实际上,即使适合自动测试的软件项目,如果不能恰当地应用自动化测试,其投资的成本会远远高于手工测试,即自动测试存在一定的风险性[4]。

Douglas Hoffman 的投资回报(ROI)分析模型指出[5],由于自动测试要编写测试程序,因此一次自动测试成本要远大于一次手工测试,自动测试成本是在反复回归测试中得到回收的。但如果考虑回归测试时测试程序的维护成本,则反复回归测试后自动测试成本将不一定会小于手工测试,自动测试风险发生。因此自动测试风险发生与否,很大程度上取决于每次回归测试时测试程序的维护成本[4]。

随着 Boehm"软件工程经济学"的推广[6],人们已经认识到软件系统维护的代价,但对于自动化测试程序的维护却并不为许多测试者重视,如 Douglas Hoffman 在定量分析自动测试的投资回报过程中就忽视了测试程序的维护代价[5]; Fewster 和 Graham 通过实例说明以增量模式开发的软件,很适合利用自动测试,但在整个论述过程中也没有考虑测试程序的维护代价[7];甚至一些知名软件公司一开始也盲目投资自动化测试,忽视了测试程序的维护代价[8];文献[4]虽然讨论了测试程序的维护代价,但却没有分析测试程序

收稿日期:2008-02-28
基金项目:国家自然科学基金(70471046);教育部博士点基金(20040359004)
作者简介:阚红星(1972-),男,安徽肥东人,博士研究生,研究方向为软件工程、自动化测试;马溪骏,教授,研究方向为管理信息系统、智能决策支持系统和知识发现。

维护的原因和过程。实际上,测试脚本是交互应用或部分非交互应用的测试自动化中必要的组成部分,它是由脚本语言编写的,因此自动化测试是一种编程测试,测试程序需要工程化,更需要维护。

文中从一个"画板"实例出发,研究了它的升级和自动回归测试过程。研究以 AbstractFigure 类为范例,详细介绍了它和相应测试程序因升级而造成的修改和变化。然后利用 COCOMO Ⅱ度量方法,对测试程序和被测程序的维护代价做一个比较,从理论上说明了测试程序和被测程序一样,需要反复维护才能进行回归测试。通过实例和一定的理论分析,使测试人员清楚地认识到自动回归测试测试程序和应用程序一样,在每次回归测试之前都需要维护,防止盲目进行自动化测试。

1 Drawlet 画板

Drawlet 画板是一个 Java 版的程序,由 Beck 和 Cunningham 共同开发[9]。画板具有可扩展性,可根据用户的需要增加或减少绘画功能。文中研究的应用程序叫 SimpleApplet,它是 Drawlet 类库的一部分。SimpleApplet 可以通过 Web 浏览器运行,它支持直线、矩形、自由曲线、圆角巨型、多边型、椭圆等基本图形的绘画,同时还配有调色板、填充样式、文本输入等功能。SimpleApplet 的顶层 UML 类图如图 1 所示。

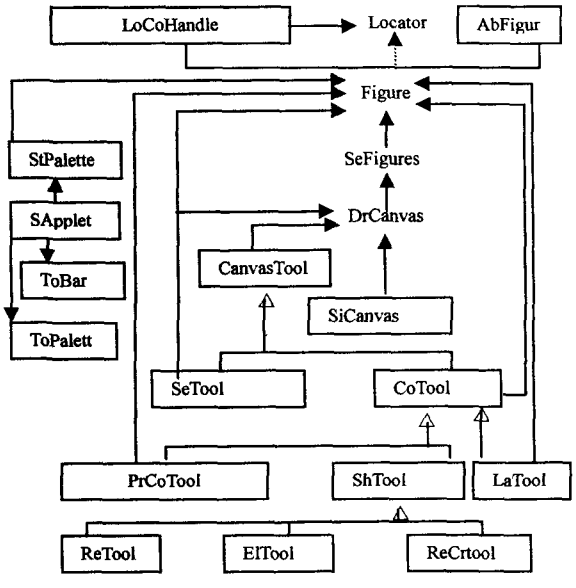

图 1 升级前 SimpleApplet 顶层 UML 类图

2 Drawlet 升级和自动测试程序的修改

2.1 画板功能升级

Rajlich 和 Gosavi 为了研究软件升级后代码所发生的改变量,他们给画板程序升级,增加了权限功能,即让用户拥有创造、修改、移动自己所画图形的权利,而这一权限别的用户不拥有[10]。画板增加权限后,运行新版 SimpleApplet,会出现一个会话框,要求用户通过账号和密码登录,或注册一个新用户。用户登录后在画布上所画的任何图形,别的用户不能更改和移动;会话框还具有侦听功能,侦听用户是否按了某个按钮,然后作相应的处理。升级后的 SimpleApplet 的顶层 UML 类图如图 2 所示。

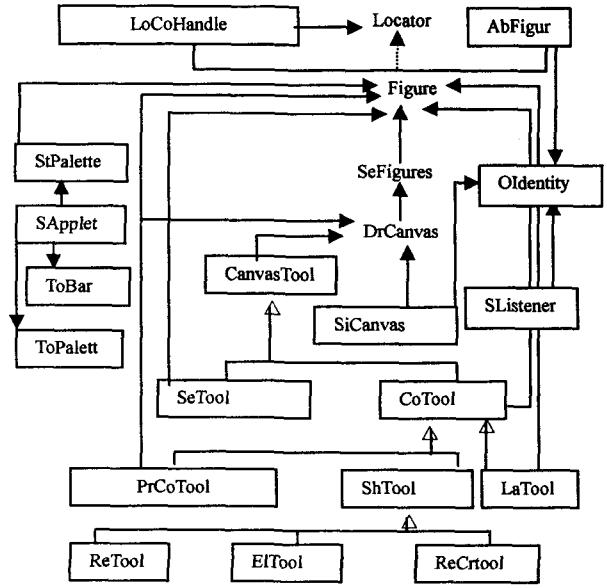

图 2 升级后 SimpleApplet 顶层 UML 类图

和图 1 比较,图 2 中出现了两个最新增加的类:OwnerIdentity 和 SimpleListener。在类 OwnerIdentity 中可实现权限设计,包括一些基本用户数据,如用户 ID 和用户姓名等,用户可通过会话框修改用户姓名和 ID,也可通过会话框侦听,侦听用户是否按了某个按钮,然后作相应的处理,侦听是在类 SimpleListener 中实现的。

2.2 画板类的修改

由上分析可知,升级后的系统要在抽象的 AbstractFigure 类中增加 3 个新的公共方法,分别是:setFigureOwner(), int getFigureOwnerID()和 String getFigureOwnerName()。setFigureOwner()方法设置用户 ID 和用户姓名,getFigureOwnerID()方法读取用户 ID,getFigureOwnerName()方法读取用户姓名。增加了新方法后,则要对 move(…)和 translate(…)方法进行修改,增加参数:securemove(…, int FigureOwnerID), securetranslate(…, int FigureOwnerID)。同样调用 move 和 translate 方法的函数也要进行修改,增加相应的参数。

就 AbstractFigure 这一个类来说,代码修改的大致情况如下:

```
public class AbstractFigure
{
⋮
private int FFigureOwnerID; //新增加的成员变量
private String FFigureOwnerName;// 新增加的成员变量
public setFigureOwner(int FigureOwnerID, String FigureOwner-
Name)
{
FFigureOwnerID=FigureOwnerID;
FFigureOwnerName=FigureOwnerName;
} //新增加的方法
public int getFigureOwnerID()
{
Return FFigureOwnerID;
} //新增加的方法
public String getFigureOwnerName()
{
Return FFigureOwnerName;
} //新增加的方法
public securemove(…,int FigureOwnerID)
{
⋮
If FFigureOwnerID=FigureOwnerID
Then …
⋮
} //修改的方法
public securetranslate(…,int FigureOwnerID)
{
⋮
If FFigureOwnerID=FigureOwnerID
Then …
⋮
} //修改的方法
⋮
}
```

上面的代码只反映了类 AbstractFigure 修改情况，与老版本的 AbstractFigure 类相比，共修改了 10 行代码。实际上，在升级中还要修改与 AbstractFigure 相关的类，因为类在互相调用时会传播类的变化，如从类 A 传递信息给类 B 要通过类 X，当类 A 发生变化时，类 B 和类 X 都要发生改变[10]。因此 SimpleApplet 的升级将导致大量代码的修改。

2.3　画板测试程序的修改

应用程序升级后要进行回归测试，以保证它们经过修改后仍能正确运行。文中的自动测试工具采用的是 JUint，它是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架。

根据 Mats Skoglund 和 Per Runeson 的研究[8]，自动回归测试的过程可分为四个阶段，分别是：环境设置、编译配置、执行单元测试、执行功能测试。在任一阶段，如果测试程序执行不成功的话都要对其进行修改。对本例来说，由于软件的升级并没有导致运行环境的改变，所以测试程序的环境设置不需要修改，而在其他几个阶段测试程序都要进行一定的修改。就 AbstractFigure 类来说，其升级后如要对它进行自动测试，JUnit 测试代码要做如下修改：

```
public class TestSample extends TestCase
{
⋮
public void testsetFigureOwner()
{
AbstractFigure Figure=new AbstractFigure();
result=Figure.setFigureOwnerID(12345, kan);
assertTrue(result);
} //新增加类的测试
public void testgetFigureOwnerID()
{
AbstractFigure Figure=new AbstractFigure();
result=Figure.getFigureOwnerID(12345);
assertTrue(result);
} //新增加类的测试
public void testgetFigureOwnerName()
{
AbstractFigure Figure=new AbstractFigure();
result=Figure.getFigureOwnerName(12345);
assertTrue(result);
} //新增加类的测试
public void testsecuremove()
{
⋮
AbstractFigure Figure=new AbstractFigure();
result=Figure.getFigureOwnerID();
assertTrue(result); //新增加参数的测试
⋮
}
public void testsecuretranslate()
{
⋮
AbstractFigure Figure=new AbstractFigure();
result=Figure.getFigureOwnerName();
assertTrue(result); //新增加参数的测试
⋮
}
⋮
}
```

由修改情况看，就 AbstractFigure 类来说，JUnit 测试程序与回归测试之前相比，增加了 15 行代码。

3 被测试程序和测试程维护代价的比较

根据 COCOMO Ⅱ[6]软件维护量的计算方法可得到令某程序总的开发成本为 V,维护成本为 C,则有:

$$C = V \times \mathrm{MCF} \times \mathrm{MAF} \tag{1}$$

其中,MCF 为维护改变系数(Maintenance Change Factor),表示程序在每次维护中代码所发生的变化;MAF 为维护调整系数(Maintenance Adjust Factor),MAF 可定义如下:

$$\mathrm{MAF} = 1 + (\frac{\mathrm{SU}}{100} \times \mathrm{UNFM}) \tag{2}$$

SU 为软件理解增量,它由 5 个等级组成,每个等级有一个确定的值,如表 1 所示。

表 1 SU 的不同等级以及相应取值

	很低	低	标准	高	很高
结构	低内聚,高耦合,面条式代码	偏低内聚,高耦合	结构十分良好,存在一些薄弱环节	高内聚,低耦合	模块性很强,信息隐藏于数据/控制结构中
应用清晰	程序与应用不匹配	程序与应用有某种相关	程序与应用中度相关	程序与应用高度相关	程序与应用清晰匹配
自描述	代码难于理解,文档丢失难于理解或陈旧	存在一些代码注释和标题,有一些有价值的文档	中等水平的代码注释、标题、文档	较好的代码注释和标题,有用的文档,但存在一些薄弱环节	代码能自描述,文档是最新的,组织良好,并包含设计原理
	50	40	30	20	10

UNFM 为程序员不熟悉性等级,根据程序员的熟悉程度取不同的值,如表 2 所示。

表 2 程序员不熟悉性(UNFM)的等级量表

UNFM 增量	不熟悉性等级
0.0	完全熟悉
0.2	大部分熟悉
0.4	部分熟悉
0.6	有点熟悉
0.8	大部分不熟悉
1.0	完全不熟悉

由公式(1)、(2)和表 1、表 2 知,程序的维护成本是由 MCF,SU,UNFM 三个参数决定的。就本例来说,无论是测试程序还是被测试程序,其软件理解增量 SU,程序员不熟悉性 UNFM 都应该是相同的,因此,维护中代码所发生的变化 MCF 决定了它们各自维护代价的大小。

令 V_1 为测试程序在维护前的代码行数,M_1 为测试程序维护后修改的代码行数,MCF_1 为测试程序代码修改百分比;令 V_2 为被测试程序在维护前(升级前)的代码行数,M_2 为被测试程序升级后修改的代码行数,MCF_2 为被测试程序代码修改百分比;令 β 为测试程序和被测程序维护代价的比值,则根据公式(1) 和公式(2) 有:

$$\beta = \frac{\mathrm{MCF}_1 \times (1 + (\frac{\mathrm{SU}}{100} \times \mathrm{UNFM})) \times V_1}{\mathrm{MCF}_2 \times (1 + (\frac{\mathrm{SU}}{100} \times \mathrm{UNFM})) \times V_2} = \frac{\frac{M_1}{V_1} \times V_1}{\frac{M_2}{V_2} \times V_2} = \frac{M_1}{M_2} \tag{3}$$

由实例分析可知,对类 AbstractFigure 来说,$M_1 = 15$,$M_2 = 10$,利用公式(3) 可得测试程序的维护代价是被测试程序的 1.5 倍。

4 结束语

投资自动测试不是一劳永逸的,当一个系统升级后,它的自动测试程序必须要经过维护,然后才能进行回归测试,这一点在投资自动测试之前必须有清醒认识,否则可能会使得自动化测试的投资永远得不到回收。

文中首先通过一个画板程序的实例说明了其升级和回归测试的维护过程,通过实践证明自动测试的维护代价是不可忽略的,然后又利用 COCOMO Ⅱ 度量方法,从理论上说明了测试程序的维护代价和被测试程序一样,是真实存在的。虽然只对画板程序的一部分 AbstractFigure 类进行了详细的分析,但它足以说明测试程序需要维护才能进行回归测试。

文中对画板的升级是个极端的例子,因为升级改变了原来程序的设计结构,所以无论是对测试程序还是被测程序来说,都造成了大量代码的修改,这也从另一方面说明软件设计的重要性,良好的结构设计不但有利于系统本身的维护,也有利于自动测试程序的维护。

参考文献:

[1] Stobie K. Too darned big to test[J]. Queue, 2005,3(1):30 - 37.

[2] Berner S, Weber R, Keller R. Observations and lessons learned from automated testing[C]//Proceedings of the 27th International Conference on Software Engineering. Missouri, USA: IEEE Press, 2005:571 - 579.

[3] Jr Brooks F P. No silver bullet essence and accidents of software engineering[J]. Computer, 1987,20(4):10 - 19.

[4] 杨善林,阚红星,一种软件自动测试成本估算控制模型[J].电子学报,2007,35(8):1588 - 1591.

[5] Hoffman D. Cost benefits ananlysis of test automation[DB/OL]. 1999 - 04 - 16[2007 - 09 - 17]. http://www.softwarequalitymethods. com/Papers/Star99% 20model% 20Paper. pdf.

(下转第 54 页)

档4326个,测试用例6598个,采用了这套同步技术后,其文档和测试用例的生存周期都有了明显的提高。

如图1和图2显示了在Elastos系统研发过程中测试用例和文档生存周期。可以看到,由于恰当运用了同步技术,测试用例和文档生存周期都有所延长,这样无疑可以大大节省软件开发中财力和物力的投入,缩短软件开发时间。

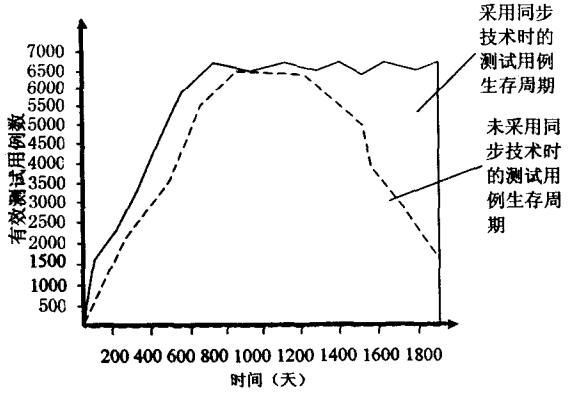

图1 测试用例生存周期的变化

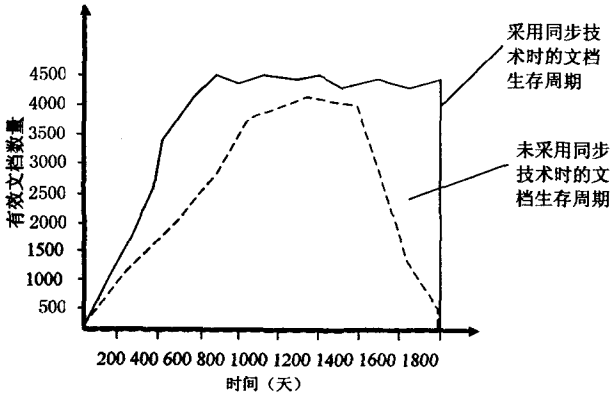

图2 文档生成周期的变化

4 结束语

程序、文档和测试用例的同步工作也是以程序的设计和修改为中心引发同步修改文档、测试用例的同步工作方式。三者通过包括文档同步工具和Bug管理系统在内的诸多技术来进行同步。这个同步机制可以通过图3来表示(箭头中斜体字表示引发同步的事件;非斜体字表示实现同步的同步机制和解决方案)。

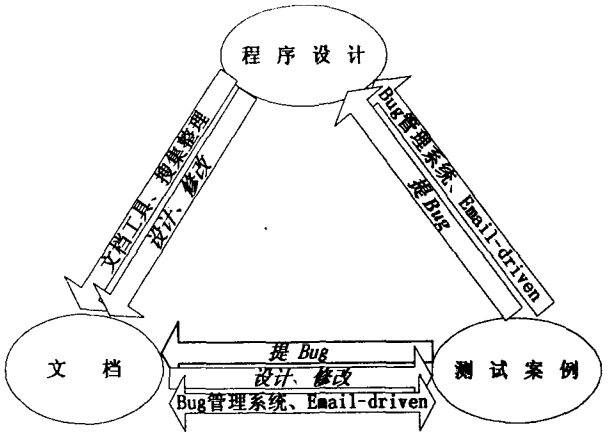

图3 程序、文档和测试用例的同步机制和管理系统

通过这些同步机制和管理方案,可以比较及时地保证系统软件研发中程序、文档和测试用例的同步。在程序发生变化的情况下,可以通过文档自动生成工具准确地同步出原始文档资料,经过文档人员将这些原始资料形成文档系统后,由测试人员开发测试用例对系统软件进行测试,验证文档和程序的正确性,如有错误再通过Bug管理系统进行文档和程序的修改和同步。这个同步过程能够使程序、文档和测试用例三者相互完善,步步同步,最终使系统软件达到成熟,呈现给用户和开发人员。

参考文献:

[1] Sommerville I. Software Engineering[M]. 北京:机械工业出版社,2003.

[2] 栾 跃. 软件开发项目管理[M]. 上海:上海交通大学出版社,2005.

[3] 田晓辉,戴金龙,沈雪芳. 如何使开发文档臻于完善[DB/OL]. 2008. http://www.uml.org.cn/bzgf/bzgf.asp.

[4] Doxygen usage[DB/OL]. 2008. http://www.stack.nl/~dimitri/doxygen/doxygen-usage.html.

[5] 孟 岩,刘振飞. Bug管理的经验和实践(中)[J]. 程序员,2005(2):38-42.

(上接第50页)

[6] Boehm B. COCOMO II model definition manual[DB/OL]. [2007-09-17]. http://sunset.usc.edu/research/COCOMOII/Cost Estimation.

[7] Fewster M, Graham D. Software test automation[M]. Boston, MA: Addison-Wesley, 1999:143-167.

[8] Skoglund M, Runeson P. A case study on testware maintenance and change strategies in system evolution[DB/OL]. 2004-11-14[2007-09-18]. http://www.ieeexplore.ieee.org/iel5/9383/29793/01357831.pdf.

[9] RoleModel Software, Inc. Drawlets[DB/OL]. [2007-09-17]. http://www.rolemodelsoftware.com/drawlets.

[10] Rajlich V, Gosavi P. A case study of unanticipated incremental change[C]//Proceedings of the International Conference on Software Maintenance (ICSM'02). Los Alamitos, CA, USA: Computer Society, 2002:359-368.