УДК 004.415.532

О.А. РЕМІННИЙ
Вінницький національний технічний університет

МОДЕЛЬ ПРОЦЕСУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ КОРИСТУВАЦЬКИХ ІНТЕРФЕЙСІВ В УМОВАХ БАГАТОПРОДУКТОВИХ КОМПАНІЙ

В даній роботі описано модель розробки та підтримки рішення автоматизованого тестування користувацьких інтерфейсів в умовах багато продуктових компаній. На основі кібернетичного підходу побудована модель зі зворотнім зв'язком. Визначено основні збудження, які впливають на розробку рішення автоматизованого тестування. Формально визначений критерій покращення якості при розробці таких рішень автоматизованого тестування

Ключові слова: Автоматизоване тестування, користувацький інтерфейс, кібернетична модель, покращення якості

O.A. REMINNYI
Vinnytsianationaltechnicaluniversity

PROCESS MODEL OF AUTOMATEDGUI TESTING IN MULTIPRODUCT COMPANIES

Abstract –In this paper we describe the process model for development and support of GUI automated testing solutions in terms of multiproduct companies. Based on cybernetic approach a model is built with the feedback. The main influences that affect the development of automated testing solutions are defined.

The concept of reusable blocks is introduced as CodeR criterion. For the first time a complex criterion ProcessImRate was formally defined to track and measureprocess quality improvement, which would allow assessing some of the procedural design decisions in the context of the time.

Keywords: Automated Testing, User Interface, cybernetic model, improving the quality

Вступ

Напевно найскладнішим рішенням впровадження автоматизованого тестування в компанії є питання чи буде воно успішним і яким буде його результат з точки зору збереження часу та коштів відносно зменшення часу на ручне тестування.

З плином часу програмний продукт стає більш складним у зв'язку із збільшенням кількості рядків коду через додавання нових функцій, виправлення існуючих помилок, і т.д. Також задачі потрібно робити в більш короткі терміни та меншою кількістю людей. Складність з плином часу буде мати тенденцію до зниження тестового покриття і в кінцевому рахунку впливає на якість продукту. Інші фактори також можуть впливати на загальну вартість продукту і час нових релізів програмного забезпечення.

Відповідно до поставлених варіацій цілей автоматизованого тестування підходять різні практики імплементації автоматизованого тестування, що в основному впливає на якість самих автоматизованих тестів. Тому актуальним є визначення певної моделі розробки рішень автоматизованого тестування, яка б дозволяла аналізувати як процес загалом, так і його певні етапи.

Метою роботи є покращення якостіавтоматизованого тестування. Відповідно в рамках даної статті *задачею* є опис моделі, яка б дозволила формально описати процес розробки рішення автоматизованого тестування, а також дозволила б визначити критерії покращення тестування.

Процес отримання інформації про якість системи

Схематично, поняття якості продукту в певний момент часу можна проаналізувати, базуючись на результатах виконання всіх тестових сценаріїв для системи. Відповідно часові затрати по отриманню можна розкласти на блоки автоматизованого та ручного виконання (Рис. 1). Початковий набір тестових сценаріїв виконується або вручну, або автоматично.

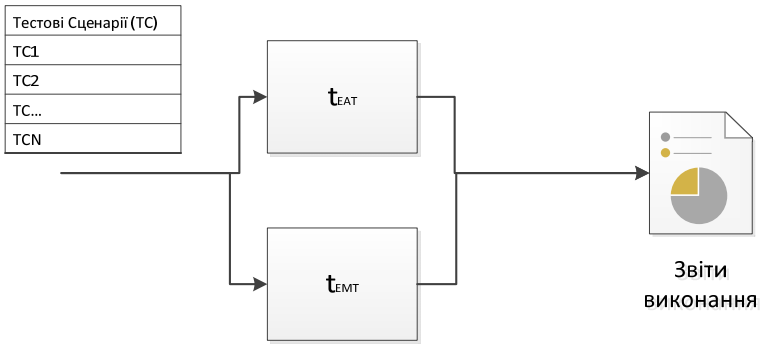


Рис. 1. Процес отримання інформації про якість системи від ручного та автоматизованого тестування

Множину всіх тестових сценаріїв (S) можна виразити як

$$S = A \cup M \,, \tag{1}$$

де A– множина автоматизованих сценаріїв,

M– множина неавтоматизованих сценаріїв, при чому для окремого тестового сценарію справедливе твердження $A(TC_i) \cap M(TC_i) = \varnothing$, так як автоматизовані сценарії немає сенсу виконувати в ручному режимі.

При такій послідовності виконання, якість системи можна виразити як залежність отримання актуальної інформації про якість (*QualityInformation*) від часу:

$$QualityInformation(t) = < t_{EMT}, t_{EAT}, Automation(A) > , \tag{2}$$

де t_{EAT}– час на виконання автоматизованих сценаріїв,

t_{EMT}– час на виконання неавтоматизованих сценаріїв,

Automation(A) - об'єм робіт по автоматизації та підтримці тестових сценаріїв.

Часові затрати на отримання кінцевої достовірної інформації про якість системи є критичними для більшості проектів з гнучкими процесами розробки [1].

Фактор часу в даній схемі є критичним. На Рис.2 ілюструється життєвий цикл проекту який було випущено в реліз та який з часом продовжують розробляти.

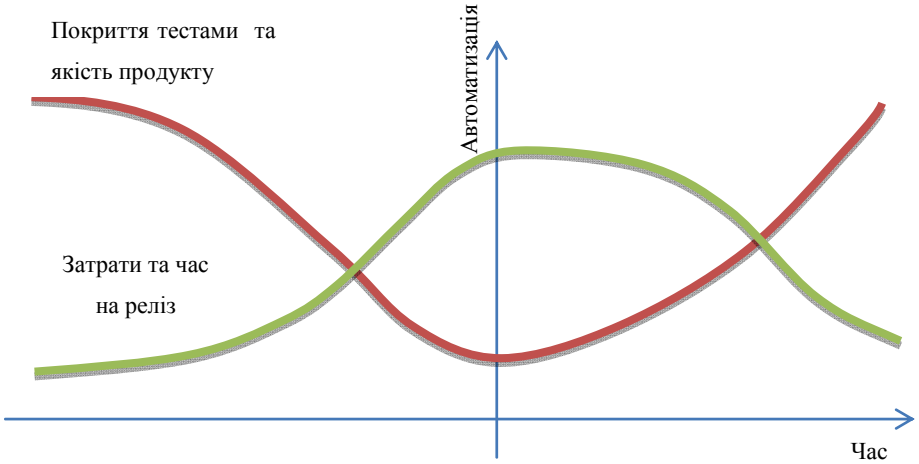


Рис.2. Витрати на тестування до та після введення автоматизації

За ринковими показниками [2], виконання одного тестового сценарію при ручному та автоматизованому тестуванні кардинально відрізняється тобто $t_{EMT} \gg t_{EAT}$. При будь-яких процесах, впроваджених в компанії, час на проведення ручного тестування сценаріїв, буде приблизно однаковим. В ідеалістичному випадку, коли $A \to S$, тоді $t_{EMT} \to 0$. Відповідно до цих тверджень, можна переписати (2) у вигляді

$$QualityInformation(t) = < Automation(S) > . \tag{3}$$

В таких випадках, основним критерієм отримання якісної інформації про продукт в певний момент часу буде залежати лише від часових затрат на автоматизацію сценаріїв.

Використовуючи кібернетичний підхід, а точніше теорію автоматизованого управління [3], можна змоделювати наступний контур (Рис. 3), який дозволяє описати об'єм затрат на автоматизацію тестових сценаріїв.

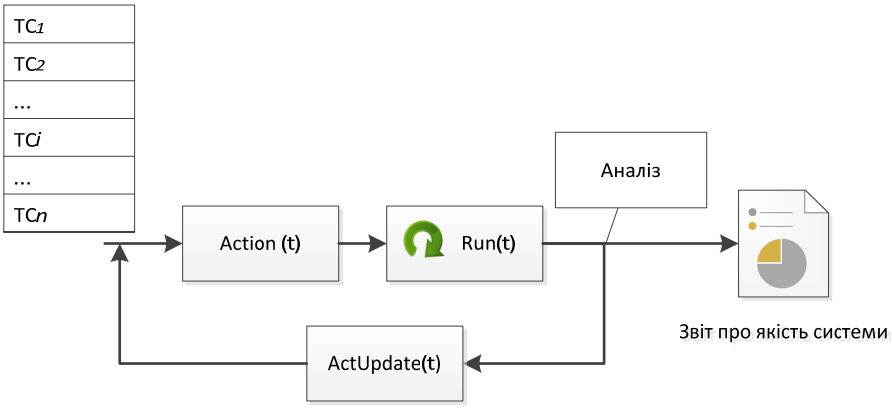


Рис. 3. Цикл розробки та підтримки рішення автоматизованого тестування

Рішення автоматизованого тестування формує собою набір сценаріїв і відповідає дії (*Action (t)*),

спрямованій на систему, що тестується. Періодичні запуски (*Run(t)*) надають програмні результати виконання тестів у вигляді множини результатів *RunRes*:

$$RunRes = ResPass \cup ResFail_{regression} \cup ResFail_{mod}, \quad (4)$$

де *ResPass* – позитивні результати виконання,

$ResFail_{mod}$ – негативні результати виконання, спричинені коректними змінами в системі, що тестується.

$ResFail_{regression}$ – негативні результати виконання, спричинені порушенням в системі, що тестується, пов'язані з додаванням нового функціоналу.

Виявлення $ResFail_{regression}$ є індикатором недопустимого функціонування системи. Подальшим кроком є корегування програмного коду системи, що тестується. $ResFail_{mod}$ є так званим *FalsePositive*[4] індикатором, тобто не є індикатором реальних помилок в тестованій системі. Відповідно, $ResFail_{mod}$ вимагає лише модифікації самого тестового рішення, щоб привести рішення до стану, відповідного новій версії системи, що тестується. *ActUpdate(t)* – це дія, спрямована на модифікацію існуючого тестового рішення з метою усунення $ResFail_{mod}$, тобто є сигналом корекції для *Action(t)*.

Відповідно до визначення, $Run(t) = t_{EAT}$. Тоді беручи до уваги (2) та процес, проілюстрований на Рис. 3, справедливим є наступне твердження:

$$QualityInformation(t) =< Action(t), \quad ActUpdate(t) >, \quad (5)$$

звідки стає очевидним залежність

$$Automation(S) =< Action(t), \quad ActUpdate(t) >. \quad (6)$$

Корпоративні рішення автоматизованого тестування

Різні підходи до автоматизації, описані в першому розділі, дуже широко розмивають спектр якості рішень автоматизованого тестування. Окреслимо основні поняття рішення, яке ми будемо розглядати далі як ідеалізоване. Оскільки в даній роботі розглядається задача підвищення якості рішення автоматизованого тестування саме через зменшення часу на його підтримку, розглянемо від яких критеріїв залежать ці затрати. В першу чергу, при підтримці будь – якого програмного рішення, виникає проблема дублювання коду [5, 6]. Відповідно, справедливим є наступне твердження:

$$ActUpdate(t) = f(CodeD), \quad (7)$$

де CodeD – явище дублювання коду [7,8] (використання одного і того ж коду у різних місцях через копіювання).

Проаналізуємо загальну задачу автоматизованого тестування більш детально. Припустимо, що у нас є складний програмний додаток з багатим інтерфейсом користувача і безліччю елементів управління. Все це розташовано на двох формах користувацького інтерфейсу. Той факт, що програмний додаток є складним, може означати, що він може мати десятки тестових сценаріїв для покриття функціоналу. Скажімо, є 50 тестів, а для більшої наочності прикладу нашою системою, що тестується, є веб - пошта Gmail. Однією з двох форм додатку є екран входу в систему. Функціонал логіну (Рис.4) використовується у всіх 50-ти тестах (для того, щоб дістатися до другої форми програмного додатку, в першу чергу потрібно авёторизуватись в додатку).

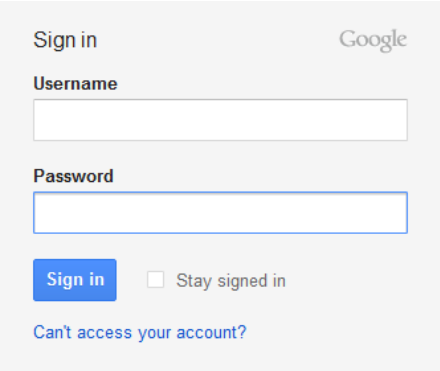


Рис.4. Форма логіну до пошти Google

Припустимо, що щось змінилося в інтерфейсі цієї логін форми. Наприклад для нашого конкретного випадку, Gmail тепер вимагає код підтвердження CAPTCHA при кожному логіні (Рис.5).

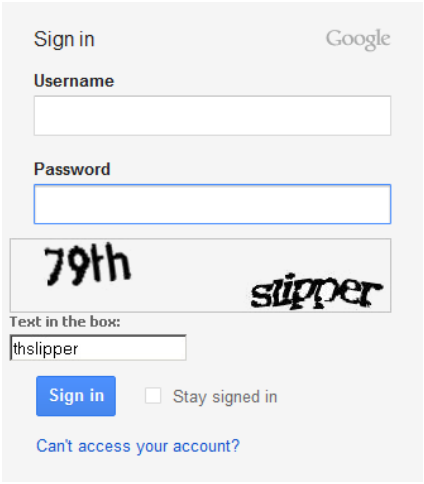


Рис.5. Форма логіну до пошти Google з CAPTCHA підтвердженням

Це означає, що кожен тест з 50-ти тепер має бути оновлений відповідно до нового робочого процесу входу в систему.

Розглянемо проблемні задачі, пов'язані з досягненням цієї цілі. Перш за все, це проблема дублювання коду [9]. Явище дублювання коду присутнє у будь яких рішеннях автоматизованого тестування. Існує багато різних методів знаходження дубльованих блоків коду в файлах вихідного коду [10,11]. На Рис.6 зображена залежність кількості часу, який потрібно витратити на відновлення рішення автоматизованого тестування в залежності від того, скільки змін спричинили до падіння тестів, а також від того, як часто повторно використовуються блоки, які спричинили падіння. Відповідна залежність є геометричною.

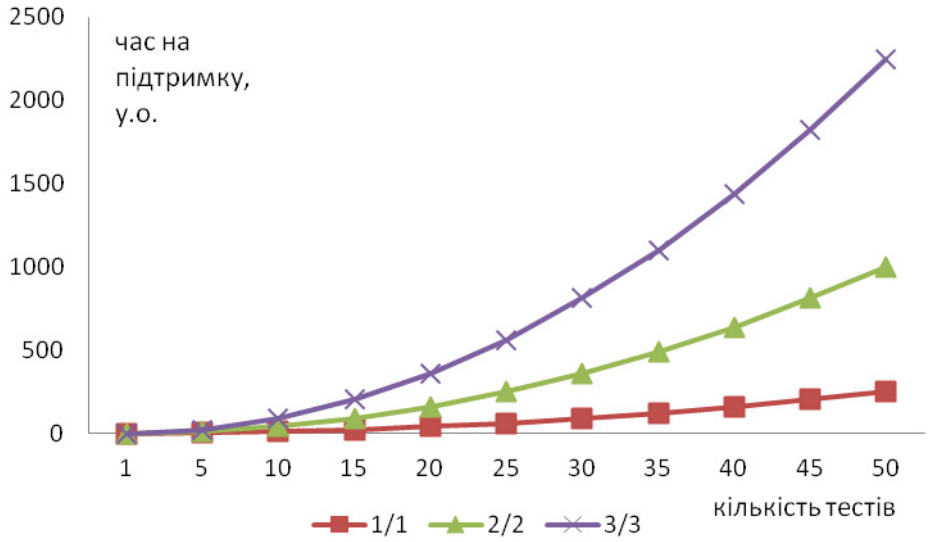


Рис.6. Час на підтримку тестового рішення при записі-відтворенні, в залежності від кількості тестів, кількості падінь/кількості перевикористань контролів

Дублювання коду при записі – відтворенні тестів в описаному випадку досягло би свого максимуму, і практично через незначну зміну потрібно було би робити перезапис всіх тестових сценаріїв.

Але загалом було б логічно, що оновлення потребує лише одна частину коду, що постійно повторно використовується. Використання наступних патернів автоматизованого тестування направлені на покращення якості коду (з точки зору затрат часу на підтримку та виправлення помилок) [12, 13, 14]:

- Багатошарова архітектура рішення;
- Мета – фреймворк;
- Об'єкт сторінки;
- Бізнес метод.

Як приклад, розглянемо бізнес метод *Login()*, який приймає параметри ім'я користувача і пароль. Нам потрібно було б оновити лише код цього функціонального методу, щоб охопити всі випадки, пов'язані з цією зміною.

Відповідно, написання коректного методу *Login()* вже є більше питанням аналізу початкових тестових сценаріїв, а таку характеристику досить важко оцінити. Однак її можна замінити, використавши показник повторного використання блоку коду *CodeR* (що є протилежним явищем відносно дублювання). Розглянемо вже описану в підрозділі 0 задачу з введенням додаткових ситуативних умов (Таблиця.1).

Таблиця 1

Приклад задачі визначення часу на підтримку рішення автоматизованого тестування

Дано	Приклад 1	Приклад 2
Кількість автоматизованих сценаріїв, N_{TC}	*50*	
Кількість бізнес методів, які спричинили падіння тестових сценаріїв, BM_F	*1*	*50*
Значення повторного використання методу, який спричинив падіння тесту, CodeR	*CodeR->Max, 50*	*CodeR->Min, 1*
Середній час на аналіз помилки падіння, tFAn	$tFAn_i$	
Продуктивність розробника	$Prod_1$	
Значення дублювання коду в рішенні автоматизованого тестування, CodeD	$CodeD_1$	

Приведений приклад наочно ілюструє, наскільки важливим є цей критерій. У випадку високого значення повторного використання коду (приклад 1), зміна лише одного бізнес методу дозволить відразу наладити роботу всього рішення автоматизованого тестування. В іншому випадку (приклад 2), розбиття коду на бізнес методи просто забезпечить модульність коду. Враховуючи вищеописані приклади, а також (7), виходить:

$$ActUpdate(t) =< CodeD, \frac{1}{CodeR} > , \tag{8}$$

Модифікуємо схему на Рис. 3 використовуючи вищенаведений опис та виключаючи запис/відтворення з набору підходів, які можуть допомагати в імплементації, отримуємо наступне (Рис.7):

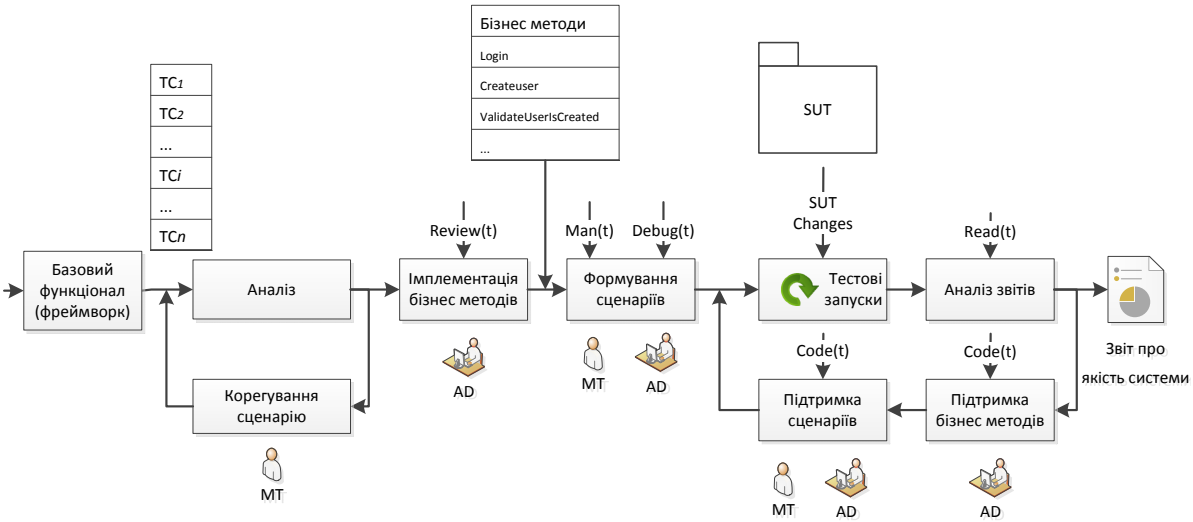


Рис.7. Кібернетичний підхід до процесу отримання інформації про якість системи. MT (тестувальник) та AD (розробник автоматизації) – ролі на етапах розробки

Відповідно до Рис.7 стає справедливим наступне:

$$\begin{cases} Action(t) =< Framework(t), Analysis(t), ImpBM(t), ImpWf(t) > \\ ActionUpdate(t) =< UpdBM(t), UpdWf(t) > \end{cases}, \tag{9}$$

де *Framework(t)* – процес імплементації базового функціоналу, придатного для повторного використання на різних проектах;

Analysis(t) – процес аналізу та адаптації тестових сценаріїв до автоматизації;

ImpBM(t) – процес імплементації бізнес методів автоматизованого рішення;

ImpWf(t) – процес побудови тестових сценаріїв з набору існуючих бізнес методів;

UpdBM(t) та *UpdWf(t)* – процес модифікації самого тестового рішення (бізнес методів та тетових сценаріїв) відповідно до виявлених в результаті тестування $ResFail_{mod}$.

Етап розробки нових тестових сценаріїв

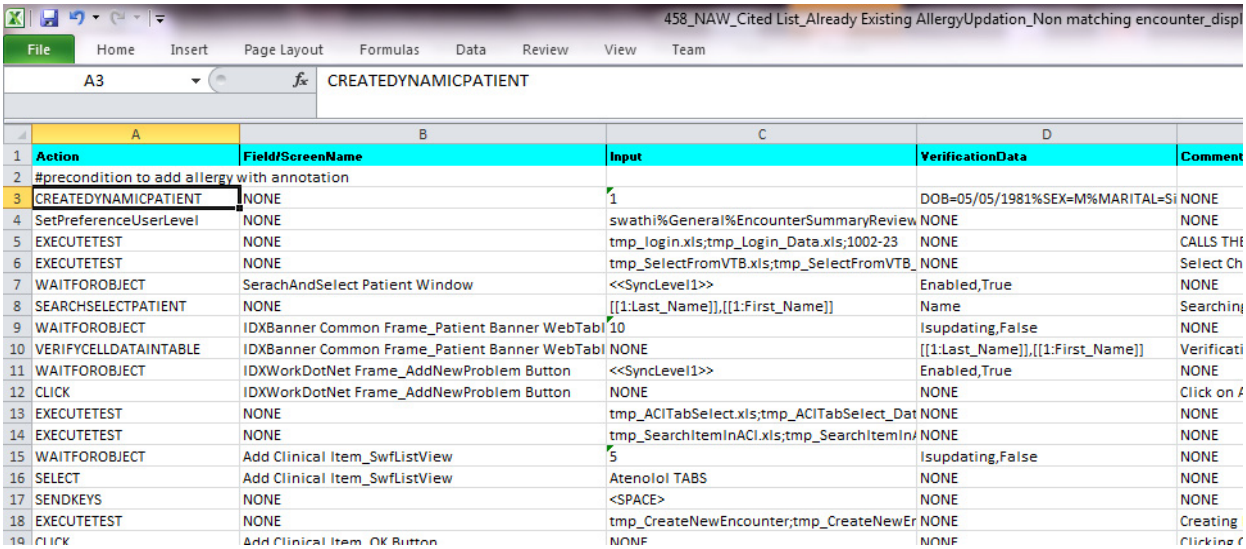
Час аналізу тестових сценаріїв залежить лише від кваліфікації тестувальника MT – $MProd$:

$$t_{An} =< MProd > \tag{10}$$

Час розробки нових бізнес методів – від процесу розробки та продуктивності розробника автоматизації при розробці *BMWork*та при налагодженні*BMDebug*:

$$t_{BM} =< BMAnalysis, BMWork, BMDebug > \tag{11}$$

Для характеристики часу розробки нових тестових сценаріїв на основі існуючих бізнес методів, більш детально розглянемо процес написання нових тестових скриптів. В залежності від специфіки проекту автоматизованого тестування, системи взаємодії з користувацьким інтерфейсом системи, яка тестується, можуть бути імплементовані різні підходи до використання бізнес методів. Вже описаний механізм все-таки потребує певного компілятора скриптів. В деяких випадках опис може бути на рівні метапрограми – наприклад в певних текстових, xml чи Excel таблицях. На Рис.8 представлено реальний приклад працюючого скрипта, який виконується певним середовищем автоматизованого тестування. Ключові слова в цьому випадку можуть бути як бізнес методами (CREATEDYNAMICPATIENT, SEARCHSELECTPATIENT), так і певними ключовими командами взаємодії з користувацьким інтерфейсом, як то CLICK для кнопок чи SENDKEYS для певних текстових полів.



458_NAW_Cited List_Already Existing AllergyUpdation_Non matching encounter_displ

File | Home | Insert | Page Layout | Formulas | Data | Review | View | Team

A3 | CREATEDYNAMICPATIENT

	A	B	C	D	
1	**Action**	**Field/ScreenName**	**Input**	**VerificationData**	**Comment**
2	#precondition to add allergy with annotation				
3	CREATEDYNAMICPATIENT	NONE	1	DOB=05/05/1981%SEX=M%MARITAL=Si	NONE
4	SetPreferenceUserLevel	NONE	swathi%General%EncounterSummaryReview	NONE	NONE
5	EXECUTETEST	NONE	tmp_login.xls;tmp_Login_Data.xls;1002-23	NONE	CALLS THE
6	EXECUTETEST	NONE	tmp_SelectFromVTB.xls;tmp_SelectFromVTB_	NONE	Select Ch
7	WAITFOROBJECT	SerachAndSelect Patient Window	<<SyncLevel1>>	Enabled,True	NONE
8	SEARCHSELECTPATIENT	NONE	[[1:Last_Name]],[[1:First_Name]]	Name	Searchin
9	WAITFOROBJECT	IDXBanner Common Frame_Patient Banner WebTabl	10	Isupdating,False	NONE
10	VERIFYCELLDATAINTABLE	IDXBanner Common Frame_Patient Banner WebTabl	NONE	[[1:Last_Name]],[[1:First_Name]]	Verificati
11	WAITFOROBJECT	IDXWorkDotNet Frame_AddNewProblem Button	<<SyncLevel1>>	Enabled,True	NONE
12	CLICK	IDXWorkDotNet Frame_AddNewProblem Button	NONE	NONE	Click on A
13	EXECUTETEST	NONE	tmp_ACITabSelect.xls;tmp_ACITabSelect_Dat	NONE	NONE
14	EXECUTETEST	NONE	tmp_SearchItemInACI.xls;tmp_SearchItemIn/	NONE	NONE
15	WAITFOROBJECT	Add Clinical Item_SwfListView	5	Isupdating,False	NONE
16	SELECT	Add Clinical Item_SwfListView	Atenolol TABS	NONE	NONE
17	SENDKEYS	NONE	<SPACE>	NONE	NONE
18	EXECUTETEST	NONE	tmp_CreateNewEncounter;tmp_CreateNewEr	NONE	Creating
19	CLICK	Add Clinical Item_OK Button	NONE	NONE	Clicking (

Рис.8. Приклад Excel скрипта

Загалом така імплементація ключовими словами є загальновідома, загальновизнана і популярна[15]. Стандартною є ситуація коли компанії розробляють власне середовище автоматизації для певних продуктів [16], і використовують саме такий підхід. Зазвичай трактується це тим, що використання ключових слів дасть змогу людям, які не розуміються на написанні програмного коду – тестувальникам – простіше писати автоматизовані скрипти.

Підхід не повністю виправданий через значну кількість недоліків. По-перше, написання ключових слів по взаємодії з користувацьким інтерфейсом системи, що тестується, через конкретний інструмент взаємодії визначає пряму залежність тестових сценаріїв від інструментів автоматизованого тестування (простіше кажучи програм, які будуть безпосередньо «нажимати кнопки» тестованої системи). Тобто, відбувається порушення принципів ізоляції шарів рішення автоматизованого тестування.

Верифікація скриптів значно ускладнюється. Потрібен додатковий запуск всього скрипта, щоб перевірити чи правильно сформована послідовність кроків. Опис тестового скрипта за допомогою стрічкових констант не дає можливості перевірити правильність написання ключових слів, помилка може бути спричинена просто орфографічною помилкою. Параметри, які передаються ключовим словам, також не мають ніякого механізму перевірки на їх кількість та тип, як то пропонується в сучасних середовищах програмування, як наприклад Microsoft VisualStudio.

Відповідно до описаних обставин, час розробки нових тестових сценаріїв сильно залежить від інструментів, які використовуються при описі. За аналогією з (11), отримуємо

$$t_{Wf} = < WfAnalysis, WfWork, WfDebug, ManualError > , \tag{12}$$

де *ManualError* – фактор ненавмисної помилки розробника тестового скрипта, зумовлений різницею написання ключових слів.

Етап підтримки існуючих тестових сценаріїв

Сумарний час t_{Upd}, що витрачається на підтримку рішення автоматизованого тестування, складається з двох основних компонентів з врахуванням (4):

$$\begin{cases} t_{Upd} = \sum tFAn_i + \sum tFix_j \\ i \in \left| ResFail_{regression} \bigcup ResFail_{Mod} \right|, \\ j \in \left| ResFail_{Mod} \right| \end{cases} \tag{13}$$

де $tFAn_i$ – часна аналіз *i*-тої помилки тестового сценарію в рішенні автоматизованого тестування. Дану величину можна виміряти емпірично і вона залежить від процесу розробки рішення автоматизованого тестування;

$tFix_j$ – часна виправлення *j*-тої помилки тестового сценарію в рішенні автоматизованого тестування.

З врахуванням (8), сумарний час виправлення помилок залежить від *CodeD* та *CodeR*, а також від числа помилок, які виникли в тестовому рішенні в результаті коректних змін в програмі $ResFail_{Mod}$:

$$t_{Fix} = \sum tFix_j =< CodeD, \frac{1}{CodeR}, ResFail_{Mod} >, \quad (14)$$

Ввівши критерій продуктивності розробника рішення автоматизованого тестування як деякий швидкісний коефіцієнт *ADProd,* отримаємо:

$$t_{Fix} = ADProd \bullet \frac{N_{ModF} \bullet CodeD}{CodeR},\ N_{ModF} =| ResFail_{Mod} | \quad (15)$$

де N_{ModF}– кількість падінь тестів, які виникли в рішенні автоматизованого тестування через коректні зміни в системі, яка тестується.

Критерій якості рішення автоматизованого тестування

Основною ціллю введення автоматизованого тестування є можливість зменшити використання людських ресурсів під час процесу тестування. Однак також очевидним є факт, що сам процес розробки автоматизованого тестування, потребує значних часових затрат. Тому доречно було би загальні часові затрати на процес, зображений на Рис.7, описати формально як часову характеристику:

$$T(P) = \begin{cases} t_{Act} = t_{An} + t_{BM} + t_{Wf} \\ t_{Upd} \\ tFAn \end{cases}, \quad (16)$$

де *T(P)* - це загальний об'єм часу витраченого на розробку та підтримку рішення автоматизованого тестування,

P – процесрозробки рішення автоматизованого тестування;

t_{Act} – час на створення початкового рішення автоматизованого тестування;

t_{An} - час на початковий аналіз тестових сценаріїв та їх корекцію, тобто підготовку до автоматизації;

t_{BM} - час написання нових бізнес методів;

t_{Wf} - час написання нових тестових сценаріїв;

t_{Upd} - час на підтримку існуючого рішення автоматизованого тестування;

Критеріям якості процесу розробки рішення автоматизованого тестування може служити сумарний час *t*, що витрачається на різних етапах розробки та підтримки рішення автоматизованого тестування. Відповідно, покращення цього показника на будь яких етапах розробки рішення автоматизованого тестування в результаті приведе до покращення самого рішення загалом. Оскільки з економічних причин недоцільно, а тому - практично неможливо, розробити одне і те ж рішення автоматизованого тестування, введемо кумулятивний відносний показник витрат часу при різних процесах розробки *ProcessImRate*:

$$ProcessImRate(P_1, P_2) = \begin{cases} ImAn = \frac{t_{An}(P_1)}{t_{An}(P_2)} \\ ImBm = \frac{t_{BM}(P_1)}{t_{BM}(P_2)} \\ ImWf = \frac{t_{Wf}(P_1)}{t_{Wf}(P_2)} \\ ImUpd = \frac{t_{Upd}(P_1)}{t_{Upd}(P_2)} \\ ImFAn = \frac{\mathrm{tFAn}(P_1)}{\mathrm{tFAn}(P_2)} \end{cases}, \quad (17)$$

де P_1 та P_2 – відповідно перший та другий процес розробки автоматизованого тестування. Префікс *Im*є похідним від слова покращення (англ. – improvement). Для наведених співвідношень, кожен частковий критерій відображатиме покращення процесу у випадку якщо його величина буде *>1*.

Висновки

В результаті проведених досліджень було визначено кібернетичну модель процесу розробки та підтримки рішення автоматизованого тестування у вигляді процесу керування зі зворотнім зв'язком. Це дало змогу формально описати процес розробки у вигляді двох складових – початкове створення рішення автоматизованого тестування та його підтримка.

Вперше запропоновано комплексний критерій виміру покращення якості процесу розробки рішення автоматизованого тестування ProcessImRate, який би дозволяв оцінювати певні процесуальні частини розробки рішення в часовому розрізі. Введено поняття повторного використання блоків коду *CodeR*.

Література

1. Shore J. The Art of Agile Development: Pragmatic guide to agile software development / Shore J. - O'Reilly Media. – 2007. – 440 p.

2. Garrett T. Implementing Automated Software Testing – Continuously Track Progress and Adjust Accordingly [Electronic resource] // Mode of access: http://www.methodsandtools.com/archive/archive.php?id=94 – Title from the screen.

3. Михалевич В. С. Словарь по кибернетике // – Киев: Главная редакция Украинской Советской Энциклопедии имени М. П. Бажана. – 1989. – 751 с.

4. Type I and type II errors [Electronic resource] // Mode of access: http://en.wikipedia.org/wiki/Type_I_and_type_II_errors– Title from the screen.

5. Kurt D., Development and Application of an Automated Source Code Maintainability Index / Welker Kurt D., Oman Paul W., Atkinson G. // – Journal of Software Maintenance: Research and Practice. – Vol3., 1997. – pp 127-159.

6. Burd E. An Initial Approach towards Measuring and Characterising Software Evolution / Burd E., Munro M. // - The Research Institute in Software Evolution. – 1999.

7. Krinke J. Identifying similar code with program dependence graphs / Krinke J.// - In Proceedings WCRE'01. IEEE Computer Society. – 2001.

8. Ducasse S. A Language Independent Approach for Detecting Duplicated Code / Ducasse S., Rieger M., Demeyer S. // Software Composition Group, University of Berne. – 2004.

9. Rieger M. Insights into System–Wide Code Duplication / Rieger M., Ducasse S., Lanza M. // Software Composition Group. – 2004.

10. Johnson J. H. Substring Matching for Clone Detection and Change Tracking / Johnson J. H. // In Proceedings of the International Conference on Software Maintence(ICSM). - 1994. - pp. 120–126.

11. Deursen A. Refactoring Test Code / Deursen A., Moonen L., Bergh A., Gerard K. // CWI Software Improvement Group. – 2004.

12. Yanhong Sun. Specification-Driven Automated Testing of GUI-Based / Sun Y., Jones E. // - ACMSE. – 2004.

13. Laukkanen P. Data-Driven and Keyword-Driven Test Automation Frameworks / Laukkanen P. // - Master's thesis, Espoo, - 2006. - 98p.

14. Reminnyi O. Functional GUI Testing Automation Patterns [Electronic resource] // InfoQ Resource. – Mode of access: www.infoq.com/articles/gui-automation-patterns – Title from the screen.

15. Котляров В. П. Основы тестирования программного обеспечения/ Котляров В. П., Коликова Т. В. – 2006 – 248с.

16. Калинов А.Я. Автоматическая генерация тестов для графического пользовательского интерфейса по UML диаграммам действий / Калинов А.Я., Косачёв А.С., Посыпкин М.А., Соколов А.А. // Труды Института системного программирования РАН. – 2004. – Т. 8.

References

1. Shore J. The Art of Agile Development: Pragmatic guide to agile software development / Shore J. - O'Reilly Media. – 2007. – 440 p.

2. Garrett T. Implementing Automated Software Testing – Continuously Track Progress and Adjust Accordingly [Electronic resource] // Mode of access: http://www.methodsandtools.com/archive/archive.php?id=94 – Title from the screen.

3. Mykhalevych V. S. Slovar' pokybernetyke // – Kyev: HlavnayaredaktsyyaUkraynskoySovetskoyEntsyklopedyyimeny M. P. Bazhana. – 1989. – 751 p.

4. Type I and type II errors [Electronic resource] // Mode of access: http://en.wikipedia.org/wiki/Type_I_and_type_II_errors– Title from the screen.

5. Kurt D., Development and Application of an Automated Source Code Maintainability Index / Welker Kurt D., Oman Paul W., Atkinson G. // – Journal of Software Maintenance: Research and Practice. – Vol3., 1997. – pp 127-159.

6. Burd E. An Initial Approach towards Measuring and Characterizing Software Evolution / Burd E., Munro M. // - The Research Institute in Software Evolution. – 1999.

7. Krinke J. Identifying similar code with program dependence graphs / Krinke J.// - In Proceedings WCRE'01. IEEE Computer Society. – 2001.

8. Ducasse S. A Language Independent Approach for Detecting Duplicated Code / Ducasse S., Rieger M., Demeyer S. // Software Composition Group, University of Berne. – 2004.

9. Rieger M. Insights into System–Wide Code Duplication / Rieger M., Ducasse S., Lanza M. // Software Composition Group. – 2004.

10. Johnson J. H. Substring Matching for Clone Detection and Change Tracking / Johnson J. H. // In Proceedings of the International Conference on Software Maintence(ICSM). - 1994. - pp. 120–126.

11. Deursen A. Refactoring Test Code / Deursen A., Moonen L., Bergh A., Gerard K. // CWI Software Improvement Group. – 2004.

12. Yanhong Sun. Specification-Driven Automated Testing of GUI-Based / Sun Y., Jones E. // - ACMSE. – 2004.

13. Laukkanen P. Data-Driven and Keyword-Driven Test Automation Frameworks / Laukkanen P. // - Master's thesis, Espoo, - 2006. - 98p.

14. Reminnyi O. Functional GUI Testing Automation Patterns [Electronic resource] // InfoQ Resource. – Mode of access: www.infoq.com/articles/gui-automation-patterns – Title from the screen.

15. Kotlyarov V. P. Osnovytestyrovanyyaprohrammnohoobespechenyya/ Kotlyarov V. P., Kolykova T. V. – 2006 – 248p.

16. KalynovA.Ya. Avtomatycheskayaheneratsyyatestovdlyahrafycheskohopol'zovatel'skohoynterfeysapo UML dyahrammamdeystvyy / KalynovA.Ya., Kosachev A.S., Posyupkyn M.A., Sokolov A.A. // TrudYnstytutasystemnohoprohrammyrovanyya RAN. – 2004. – Vol. 8.

Рецензія/Peer review : 31.12.2013 р. Надрукована/Printed :7.1.2013 р.
Рецензент: Бісікало О.В., д.т.н., проф. кафедри АІВТ ВНТУ