Софтуерна верификация

Михаил Николаев Николов
Специалност: Компютърни науки
ФН: 80527
3-ти курс
3-та група

Софтуерната верификация е дисциплина от софтуерното инженерство, чиято цел е да се гарантира, че софтуерът напълно отговаря на всички очаквани изисквания. Процесът на проверка е важна част от разработката на софтуера, която може да спести много средства и други негативни последствия от евентуални грешки или недостатъци в софтуерния продукт. Но не добре планираните тестове биха могли да костват много време, а от там и разходи за фирмата. За това подборът и комбинирането на подходящите подходи и методи за проверка е от изключително значение.

Има два основни подхода за проверка – динамична и статична верификация. Динамичната верификация е позната също като тестване и е добра за откриването на грешки, а статичната верификация е процес на проверка, дали софтуерът отговаря на поставените изисквания. При започването на един софтуерен проект стои въпросът кой от двата подхода да използваме и дали не бихме могли да ги комбинираме по някакъв начин. Еднозначен отговор няма и решението до голяма степен зависи от спецификата на задачата, която решава софтуерът и от средствата и времето с които разполага екипът разработващ програмата.

Освен този въпрос, възникват и други при избора на методите за тестване, които се използват за динамичната верификация. Обичайно процесът протича на няколко етапа от разработката на продукта и често се налага повторното провеждане на някои тестове. При по-големи и сложни проекти е трудно проследяването на зависимостите и не рядко промяна в една функционалност на приложението предизвиква грешка или неочаквано поведение в на пръв поглед напълно различна функционалност. За това е необходимо повторното провеждане на проверки и лошият подбор на тестове, тяхното дублиране или пропускането на някои ситуации могат да имат неприятни последствия, които да компрометират целия процес на верификация на софтуера. Практиката е доказала необходимостта от наличието на някои етапи от динамичната верификация, които са в пряка зависимост и приложени заедно в правилната последователност дават максимално добри резултати. Те разделят изходния код на програмата на отделни компоненти, които тестват най-напред поотделно, а в последствие и комбинирани заедно. Но освен този метод, които разглежда работата на кода, съществува и друг, наречен black-box метод, който тества поведението на софтуера без да се интересува от реализацията и логиката използвана за достигане на търсения резултат.

При избора на подход и комбинация от методи за проверка коректността на софтуера трябва да вземем предвид и предимствата и недостатъците им. Някои подходи са изключително трудоемки и изискват неоправдано много време. Други, макар и да изискват голяма инвестиция на човешки и времеви ресурс в подготовката, са гаранция за навременното откриване на грешки и елиминирането им за целия процес на разработка.

Динамична верификация

Динамичната верификация се извършва по време на изпълнението на софтуера и динамично проверява поведението му. Тази фаза е известна като тестване. Тъй като софтуерът е в ход докато динамичните инструменти правят проверката, фалшивите положителни резултати са ряд[1]кост. Въпреки това, тъй като динамичните инструменти откриват само действията, които се случват по време на наблюдението на софтуера, те могат да допуснат някои грешки, ако обхватът на теста не е адекватен. В същото време, чрез използването на наличната информация по време на изпълнение, например информацията, която е по-трудно да се извлече статично от изходния код, динамичните инструменти за проверка

1

могат да открият определени видове грешки, които са трудни за откриване със статичните средства за анализ.

В зависимост от обхвата на тестовете, можем да ги разделим на малки и големи. Към малките спадат така наречените unit тестове, които проверяват обикновено една функция или клас. Към големите тестове се причисляват следващите стъпки в процеса на динамична верификация – integration и system тестовете. Първите подлагат на проверка вече интегрирани модули, тествани по-рано, докато системните тестове изследват поведението на цялата система. След завършването на приложението се извършва проверка за приемане (acceptance testing), която проверява критериите за приемане на софтуера *[виж Приложение 1]*. От изключително значение и провеждането на regression тестове след всяка промяна по приложението, които да гарантират, че не са възникнали нови грешки нито пък са се появили отново стари.

Само изчерпателното изследване може да покаже, че програмата е без дефекти, но както знаем напълно изчерпателно тестване е практически невъзможно. Друг важен аспект, който трябва да се има предвид при изработката на тестове е, че те трябва да изследват възможностите на програмата, а не компонентите ѝ. При създаването на тестващият софтуер е добре да се ръководим от това, че типичните ситуации са най-важни и винаги са с по-голям приоритет пред граничните случаи.

За разработката на тестовите ситуации се използва black-box или white-box метода. Black-box случаите са разработени посредством анализ на потребителските и системните изисквания, а white-box случаите – чрез изследване на поведението на изходния код.

Black-box testing

Black-box testing-а е метод за тестване на софтуер, който разглежда функционалността на едно приложение без да се интересува от вътрешната му структура или начин на работа. Този метод може да се прилага за почти всяко ниво на софтуерното тестване – unit, integration, system и acceptance.

За процедурата по тестване не са необходими специални познания за вътрешната структура на приложението. Разработчикът на тестовете трябва да е наясно с това какво софтуерът трябва да прави, но не и как го прави. Например, той знае, че за дадени входни данни има точно определен резултат, но не знае как софтуерът достига до този резултат *[виж Приложение 2]*.

Тестовите варианти са изградени около спецификацията и изискванията, т.е. това което приложението трябва да прави. Тестовите ситуации обикновено се получават от външни описания на софтуера, включително спецификации, изисквания и проекти параметри. Въпреки, че използваните тестове са с предимно функционален характер, нефункционални тестове също могат да се използват. Разработчикът на тестовете избира двата входа – валиден и невалиден и определя коректния изход без никакви познания за вътрешната структура на обекта.

Едно от най-големите предимства на black-box метода е и неговата основна характеристика – разработчикът не се нуждае от познания за начина на работа на приложението, нито е нужно да се запознава с изходния код. Това позволява разработката на тестовете да върви независимо от развитието на софтуерът, тъй като не е необходим достъп до кода, а предварително известните спецификации и изисквания са напълно достатъчни. Освен това, често при по-големи проекти се използва различни технологии и по няколко езика за програмиране, което би изисквало разработчиците на тестове за са специалисти в много различни сфери, за да са в състояние да ра[2]збират кода. При black-box метода тестовете могат да се

2

изготвят от хора, които нямат никакви познания по използваните технологии, без[3] това да влияе на качеството на тяхната работа. Това позволява по-лесното сформиране на по-голям екип за разработка на тестове. Всички тези предимства правят black-box метода ефективен при тестването на големи системи.

Основен недостатък на метода е липсата на информация за реализацията на приложението прави по-трудно определянето на всички възможни входни данни, което може да доведе до тестове, покриващи само част от ситуациите, а от там и до пропускането на някои грешки. Разработчиците на тестващия софтуер са изправени пред предизвикателството да изработят тестовите варианти, което може да е особено трудно при неясни функционални спецификации. Независимата работа на екипа изготвящ тестовете от екипът разработващ приложението е нож с две остриета. Освен посочените по-горе предимства, това може да доведе до дублиране на тестове, изготвени от разработчиците и загуба на ценно време за и без това обемната работа при по-големи проекти.

White-box testing

White-box testing-ът е метод за тестване на софтуер, който изпитва вътрешната структура на приложението, за разлика от неговата функционалност (обратно на black-box testing-а). При изработката на тестове по метода white-box са необходими познания за структурата на приложението и умения по програмиране. Избират се входни данни за тестване на части от входа и се определят подходящите резултати.

White-box методът може да се прилага при unit, integration и system нивата на процеса на софтуерно тестване, но обикновено се използва предимно при unit тестовете.

Въпреки, че този метод може да разкрие много грешки или проблеми, той не може да открие неизпълнени части на спецификацията или липсващи изисквания.

White-box testing-ът е метод за проверка на приложението на ниво изходен код. Тестовите варианти се получават чрез използването на техники за дизайн като control flow testing, data flow testing, branch testing, path testing statement coverage, decision coverage и др. White-box използва тези техники и насоки, за ад създаде среда без грешки, проучвайки всяка потенциално проблемна част от кода. Тези техники са градивните елементи на white-box testing метода, чиято същност е внимателното изследване на приложението на ниво изходен код, за предотвратяване на всякакви скрити грешки по-късно. Чрез тези техники се изпитва цялостния изходен код, за да се сведат до минимум грешките и да се създаде една среда без грешки. Плюс на white-box testing-а е и възможността да разберем кой ред от изходния код се изпълнява в момента, както и да определим какъв трябва да бъде правилният резултат.

White-box методът се използва по време на unit тестовете, за да се гарантира, че кодът работи по предназначение преди да бъде интегриран с предварително тестван код. White-box testing-ът по време на unit тестовете улавя всякакви дефекти в ранна фаза и помага при дефекти възникнали по-късно, след интеграцията с останалата част от приложението, предотвратявайки грешки в по-късен етап.

На ниво intergation testing, white-box методът се използва за проверка на взаимодействието на един интерфейс с останалите. Unit тестовете се грижат всеки код, който се проверява да работи правилно в изолирана среда (сам за себе си), а integration тестовете разглеждат коректността на поведението в отворена среда чрез white-box тестове на всички взаимодействия на интерфейси, известни на програмиста.

3

White-box проверката по време на regression testing-а е повторното използване на тестовите ситуации то unit и integration тестовете. Regression тестове е необходимо да се извършват след всяка промяна по софтуера с цел да се провери дали не е довела до нови грешки или не влияе негативно върху работата на останалата част от програмата.

Основните процедури на white-box testing-а включват разбирането на изходния код в дълбочина, за да бъдем в състояние да го тестваме. Програмистът трябва да има по-задълбочени познания за работата на приложението, за да знае какви тестове да създаде, така че да постигне пълно покритие на всички възможности. Веднъж разбран, изходният код може да бъде анализиран за избор на тестови варианти.

Основните стъпки за създаването на white-box тесответе са три. Първата включва подбора на подходящи входни данни за тестовете, имайки предвид различните видове изисквания, функционални спецификации и спецификации за сигурност. Това е подготвителният етап за оформление на цялата основна информация. Втората стъпка се състои в анализ на риска за насочване на целия процес на тестване, правилно планиране на теста, изпълняването му и съобщаването на резултатите. Това е фазата на подбор на тестовите варианти, при които щателно проверяваме поведението на приложението и записваме получените резултати. При третата стъпка вече разполагаме с резултатите от проведените тестове и подготвяме окончателен доклад, който охваща подготовката за тестовете и получените резултати.

White-box тестването е един от двата най-използвани метода днес. Той има преди всичко три предимства. Познаването на кода, което е абсолютно задължително за този метод, има един положителен страничен ефект и това е възможността за по-задълбочено тестване. Неминуемо един програмист, заподнат с изходния код е в състояние да създаде тестове, покриващи много по-добре възможните обичайни и гранични ситуации, от колкото например по метода black-box, където може да разчита само на предварително обявените спецификации. White-box позволява и оптимизиране на кода чрез разкриване на скрити грешки и премахване на евентуални дефекти в ранен етап от разработката на софтуера. Освен това дава на програмиста възможност за самоанализ, защото разработчиците внимателно описват всяко ново изпълнение.

Въпреки, че white-box тестването има много предимства, то не е перфектно и има и своите недостатъци. White-box методът усложнява тестването, защото, за да може да се изпита всеки важен аспект на програмата, се изискват задълбочени познания за кода и структурата ѝ. Необходимо е програмистът да бъде с високо ниво на познания. В някои случаи не е възможно да се тестват всички съществуващи състояния на приложението и някои от тях ще останат неизпитани. Друг потенциален проблем на white-box метода е, че ако дадена функционалност е пропусната и не е реализирана, то тя няма да бъде отразена и в теста, тъй като той се прави по съществуващия код. Поради необходимостта от високо квалифицирани разработчици, които да бъдат запознати с кода, разходите по проекта и в частност за подготовката и провеждането на тестовете са значително по-големи от тези при black-box метода например.

Black-box testing vs. White-box testing

Докато white-box метода валидира вътрешната структура и работа на кода, основният фокус на black-box testing-а е валидирането на функционалните изиксвания.[4]

4

За провеждането на white-box тестове, познаването на основният програмен[5] език, използван при разработката, е от съществено значение. В днешно време софтуерните продукти са изградени от множество езици за програмиране и други технологии и задълбочени познания по всички тях са практически невъзможни. Чрез black-box метода се абстрахираме от кода и можем да се фокусираме върху поведението на приложението.

Също така софтуерните системи не се разработват като едно цяло, а като различни модули, което прави black-box метода по-подходящ за тестване на комуникацията мужду модулите (integration testing).

Когато направим промени по кода на вече комплектуван софтуерен продукт, пълната проверка на системата придобива първостепенно значение. За целта е удачно да се използват black-box regression тестове.

Въпреки това white-box методът е изключително полезен за откриване на много вътрешни грешки, които биха могли да влошат работата на системата.

Black-box и white-box тестовете могат да бъдат разделени на няколко категории. Black-box тестовете можем да разглеждаме като функционални, нефункционални и regression тестове. Функционалните тестове са свързани с функционалните изисквания на системата. Нефункционалните тестове не са свързани с тестването на специфични функции, а с нефункционалните изисквания като производителност, скалируемост, използваемост. Regression тестовете се извършват след промени по кода, надстройка или други промени, за да проверят дали новият код не е повлиял на съществуващия. От своя страна white-box тестовете се делят на unit тестове, memory leaks тесотве, white-box penetration тестове и white-box mutation тестове. Unit тестовете, както вече е казахме, подлагат на изпитание малки части от кода, като отделни функции или класове. Memory Leaks тестовете имат за цел да открият „течове“, където се губи памет, което може да бъде пагубно за приложението, особено ако се предвижда то да работи на устройства с ограничена памет. White-box penetration testing-а е метод при който разработчикът има пълна информация за изходния код на програмата, подробна информация за мрежата и за работата на приложението. Целта е да се атакува кода от „различни ъгли“ с цел разкриване заплахите за сигурността. White-box mutation тестовете често се използват за определяне на най-подходящите техники за писане на код, които да се използват при развитието на софтуера.

Unit testing

Основната цел на unit тестовете е да вземат най-малкото звено от софтуера, което може да бъде тествано самостоятелно, да го изолират от останалата част от кода и да определят дали се държи така, както се очаква от него. Всички звена се проверява поотделно преди интегрирането им в модули. След това се тестват връзките между модулите (чрез integration тестове).

Unit тестовете са се доказали като ефективен метода за откриване на голяма част от дефектите. Пропускането на unit тестовете е силно непрепоръчително. Ако решим да комбинираме няколко звена в един модул и едва тогава да проведем тестове, няма как да знаем от къде идва евентуална грешка. Тя може да е във всяко едно от звената, в няколко от тях или в комуникацията помежду им.

Unit тестовете имат своите ползи, които са ги утвърдили като неизменна част от процеса на верификация на софтуера.

Основно предимство на unit тестовете е, че проблемите се откриват много рано в процеса на разработка. Обикновено unit тестовете се разработват преди съответната част от кода, която ще бъде тествана, да бъде написана. След като

5

мине успешно теста, въпросната част от кода се счита за завършена. Същите тестове се изпълняват отново след всяка промяна по кода. Ако даден тест е неуспешен се счита, че има проблем с тествания код или със самия тест. В по-късен етап от разработката на софтуера, unit тестовете позволяват лесно проследяване на грешките. Unit тестовете предупреждават разработчиците за проблеми, преди продукта да достигне до клиента, което е от съществено значение.

Unit тестовете позволяват на програмиста, когато направи промени по кода в по-късен момент, лесно да се увери, че всичко работи коректно (например в regression тестовете). Процедурата е да се пишат тестове за всички функции и методи, така че в случай на промяна предизвикваща грешка, проблемът да бъде бързо идентифициран и поправен. Освен, че помагат на програмиста лесно да установи дали дадена част от кода функционира правилно още в процеса на разработка, unit тестовете са изключително полезни и в много по-късен етап, защото продължават да изпълняват функцията си независимо от промените в останалата част от кода *[виж Приложение 3]*.

След тестване на всички звена на програмата чрез unit тестове, тестването им заедно (integtarion testing) става много по-лесно, а евентуална грешка – по-лесно откриваема. Много прости грешки, които лесно могат до бъда открити с помощта на unit тестове, отнема много време да бъдат проследени, ако пропуснем unit тестовете и се натъкнем на тях след интегрирането, едва при integration тестовете.

Тестовете не са в състояние да открият всяка грешка в програмата. В частност това се отнася и за unit тестовете. Освен това, те по правило изпитват само функционалността на звеното, за което са предназначени. Поради това не могат да хванат грешки възникващи при интеграцията или грешки на системно ниво. Unit тестовете могат да покажат само наличието или липсата на определени грешки, но не могат да докажат пълната липса на грешки. За да се гарантира правилното поведение на софтуера във всяка ситуация и при всякакви входни данни и да се гарантира липсата на грешки е необходимо прилагането на формални методи за доказателство, че софтуерният компонент няма неочаквано поведение.

Тестването на софтуер е комбинаторен проблем. Например всяка булево твърдение изисква най-малко две изследвания – при true или false. В резултат на това често на един ред код от тествания софтуер отговарят 3-4 или повече реда в изходния код на теста. Това очевидно отнема време и инвестицията може и да не си струва усилията. Има и много проблеми, които не могат лесно да бъдат тествани във всички ситуации – например, тези които са недетерминирани или включват няколко нишки. Също така не е за пренебрегване фактът, че тестовете що могат да съдържат грешки.

Друго предизвикателство свързано със създаването на unit тестове е трудността за създаване на реалистични и полезни тестове. Необходимо е да се създадат съответните начални условия, така че частта от приложението, която се тества, да се държи като част от цялата система. Ако тези начални условия не са създадени правилно, тестът няма да изпробва кода в реалистичен контекст, което намалява стойността и точността на unit теста.

За да се получат желаните резултати от unit тестовете е необходима строга дисциплина по време на процеса на разработка на софтуера. От съществено значение е да се пазят подборни записки, не само от тестовете, които са били извършени, но също така и за всички промени, които са били направени по изходния код или друга част на софтуера.

Използването на система за контрол на версиите е от съществено значение.[6]

6

Ако по-нова версия на приложението не издържа даден тест, който преди това е[7] бил преминат успешно, системата за контрол на версиите може да представи списък на промените, което значително улеснява проследяване и откриването на проблема.

Също така е от съществено значение неуспешните тестове да се проверяват ежедневно и да се разглеждат веднага. Ако това не се внедри в работния процес на екипа, приложението няма да се развива в синхрон с unit тестовете, което може да повиши фалшивите положителни резултати и да намали ефективността на тестовете.

Integration testing

Integration testing-а е фаза от софтуерното тестване, в която отделните софтуерни модули се комплектуват и се тестват като група. Това се случва сле дunit тестването. Методът използва части от кода, които са преминали успешно unit тестовете по-рано, групира ги и ги подлага на проверка.

Целта на integration тестовете е да проверят функционалността, производителността и изискванията за надеждност заложени в дизайна на софтуера. Вече комплектуваните модули се тестват чрез black-box метода, а успехите и случаите на грешки се симулират чрез подходящи входни данни. Тестовите варианти са конструирани така, че да изпробват всички взаимоотношения между отделните модули правилно.

Integration testing-а е логично продължение ан unit tedting-а. В най-простата си форма два отрязъка от код, които вече са били тествани, са комбинирани в компонент и интерфейсът между тях се изпробва.

Има няколко подхода при integration тестовете – Big bang, Отдолу-нагоре и Отгоре-надолу.

При big bang подхода всички или повечето от разработените модули са свързани заедно, за да образуват цялостна софтуерна система или голяма част от системата, след което се провеждат integration тестовете. Методът на „големия взрив“ е много ефективен за спестяван е на време в процеса на integration testing. Въпреки това, ако тестовете и резултатите от тях не се отчитат правилно, целият процес на интеграция ще се усложни и това може да попречи на постигането на целта на integration тестовете.

Подходът Отдолу-нагоре е подход за integration тестване, при който първо се изпитват компонентите на най-ниско ниво, а след това и на по-високо ниво. Процесът се повтаря докато не тестваме компонентите на върха на йерархията. Всички модули на ниско ниво се интегрират и след това се тестват. Модулите от следващото ниво се образуват след това. Този подход е полезен, само когато всички или повечето модули на едно и също ниво на развитие са готови. Този метод помага да се определи до къде е стигнала разработката на софтуера.

Отгоре-надолу е подход, при който се изпитват първо модулите от най-високо ниво. Това позволява високо ниво на логика. Недостатък е, че модулите от по-ниско ниво се тестват сравнително късно.

Основно предимство на подхода отдолу-нагоре е по-лесното откриване на грешки. При подхода отгоре-надолу е по-лесно откриването на липсващ клон в логическата структура на приложението. Можем да обобщим предимствата и недостатъците на методите отдолу-нагоре и отгоре-надолу накратко по следния начин.

Подходът отдолу-нагоре позволява по-лесно откриване на грешки и при него не се налага да чакаме всички модули да бъдат завършени, за разлика от Big bang подхода. Въпреки това, минус е, че критичните модули, намиращи се в

7

най-високото ниво в софтуерната архитектура, се тестват последни и може да са податливи на дефекти. Като недостатък можем да отбележим и невъзможността за създаване на ранен прототип.

За сметка на това, подходът отгоре-надолу дава възможност за получаване на ранен прототип, а критичните модули се тестват с приоритет. Това позволява сериозни дефекти в дизайна да бъдат открити и отстранени първи. Недостатък е необходимостта от много разклонения при тестване и недостатъчното тестване на модулите от най-ниско ниво.

System testing

Системното тестване на софтуер са тестове проведени върху цялата система, след интегрирането на всички модули, за оценяване на съответствието на системата с изискванията. Системните тестове поподат в обхвата на black-box метода и съответно не трябва да изискват никакви познания по кода или логиката, по която е реализирано приложението.

Като правило, системните тестове приемат за вход всички интегрирани софтуерни компоненти, които успешно са преминали integration тестовете, както и самата система, интегрирана с хурдуера.

Системните тестове се извършват на цялата система в рамките на техническата спецификация и/или спецификацията на системните изисквания. System тестовете изпитват не само дизайна, но и поведението на системата.

Acceptance testing

Целта на acceptance тестовете е да се оцени съответствието на системата с изискванията на бизнеса и да се прецени дали софтуерът е готов да се представи на крайните потребители.

Обикновено за acceptance тестовете се използва black-box метода. Тестовете при приемане се извършват след системните тестове и преди софтуерният продукт да бъде пуснат в употреба.

Вътрешните acceptance тестове се извършват от членове на организацията, разработила софтуера, но които не са пряко ангажирани с проекта (разработване или тестване). Външната проверка се извършва от хора, които не са служители на фирмата. Клиентските acceptance тестове се извършват от клиенти на разработващата фирма, а потребителските – от крайните потребители на софтуера.

Regression testing

След всяка промяна по софтуера е необходимо да се извършват regression тестове. Целта им е да се гарантира, че промените нае водят до нови грешки. Една от основните причини за провеждането на regression тестове е да се определи дали промяната в една част от софтуера засяга други части.

Обикновено regression testing-а включва изпълнението на по-рано проведени тестове и проверка дали поведението на програмата не се е променила и дали по-рано поправени грешки не са се появили отново.

Опитът показва, че промените по софтуера често водят до възникването на нови или изплуването на стари грешки. Една корекция може да решава проблем в някакъв частен случай, където за пръв път е бил наблюдаван (проблемът), но не и в по-общи случаи, които могат да възникнат при ползването на софтуера. Също така, не рядко решаването на проблем в една област довежда до появата на нови грешки в друга.

Поради това, използването но regression тестове се счита за добра практика.

Най-ефективният подход при regression testing-а се основава на[8]

8

разработката на библиотека, съдържаща стандартен набор от тестове, които да се[9] изпълняват след завършване на всяка нова версия на програмата. Най-трудният аспект при изграждането на библиотеката е да се определи кои тестове да се включат. Почти задължителни са автоматичните тестове и тестове и тези включващи гранични случаи. Някои компании за разработката на софтуер включват само тестове, които действително са открили грешки. Проблемът е, че точно тези грешки са били поправени отдавна, а и не е изключено промените по софтуера да предизвикат нови грешки, там където преди това не е имало такива.

Необходимост е и периодично да се прави преглед на библиотеката, за премахване на повтарящи се или излишни тестове. Дублирането е често срещано, когато повече от един човек се занимава с разработката на тестове.

Статична верификация

Статичната верификация е анализ на компютърния софтуер, който се извършва без реално изпълнение на програмата. Статичната верификация цели да провери дали софтуерът отговаря на изискванията. Това става чрез проверка дали се спазват конвенциите при писане на код, откриване на лоши практики, изчисление на софтуерните метрики и официална проверка (формална верификация) – доказване или опровергаване на верността на алгоритъма, залегнал в основата на системата, посредством формални математически методи. Методът за проверка чрез анализ се състои в проверка чрез изследване, математически изчисления, логическа оценка и изчисления с класическите научни методи или общоприети методи използващи компютри. Анализът включва сравняване на наблюдаваните резултати от тестове с изчислени очаквани стойности в съответствие с изискванията.

Статичният анализ изследва какво прави програмата без да я изпълнява чрез внимателно изпитване на изходния код. Статичният анализ може да диагностицира нарушения на правилата и конвенциите, които са необходими за правилното изпълнение на програмата или се предпочитат за някои нефункционални качествени показатели като например поддръжка и сложност (но не и ползваемост или производителност). Също така може да диагностицира съответствие със стандартите при писане на код и най-добрите практики за програмиране и безопасно програмиране.

Статичните инструменти изследват кода без той да се изпълнява. Тъй като тези инструменти не разчитат на тестове, които да симулират изпълнение на кода те могат да бъдат изключително задълбочени. Теоретично, статичните инструменти за проверка могат да изследват целия код, включително части от него, които са рядко изпълнявани на практика. Въпреки това, тъй като софтуерът действително не работи, могат да се генерират фалшиви положителни резултати. Това означава, че може да се докладват грешки, които не могат да появят в практиката.

Статичният анализ предоставя информация, която е валидна за всички възможни изпълнения на програмата. Тъй като за провеждането на проверката не е необходимо изпълнението на програмата, то тя може да бъде извършена преди имплементацията да е напълно завършена. При една проверка могат да бъдат открити различни дефекти, но след отстраняването на един от тях е необходима

9

повторна проверка, тъй като една грешка може да прикрие друга. Проверките често откриват дефекти, които тестовете не са успели да открият. Статичната верификация може да окаже се положителна и за екипа, провеждащ проверките, тъй като процесът помага на хората да осъзнаят необходимостта от спазването на стандартите.

В същото време, ранното откриване на грешки може да спести много време и средства, понеже е много по-лесно поправянето на грешки още в процеса на разработка, отколкото след интегрирането на всички модули на програмата или след пускането ѝ в употреба.

Недостатъците на статичната верификация също не са за подценяване. Процесът е скъп и отнема време, а работата е досадна, което увеличава риска от грешки. Проверките увеличават разходите в началото на софтуерния процес, а обикновено се налага повторното им провеждане. Минус е и необходимостта екипът извършващ проверките предварително да се запознае със стандартите и насоките, които трябва да бъдат налице. Освен това представената информация не винаги е точна, тъй като тя обикновено се основа на приближение.

Подбора на подходящи методи и средства за верификация е от голямо значение за успеха на софтуерния проект като избора и комбинирането на правилните подходи може да спести много време и средства. Статичната и динамичната верификация имат своите плюсове и минуси и изборът на само един от двата подхода и пълното загърбване на другия би имал по-скоро негативен ефект. Анализът „гледа“ на софтуера по-скоро от математическа и логическа гледна точка, което позволява задълбочени изследвания на кода. Този подход не е за пренебрегване и следва да бъде включен в процеса на верификация, тъй като позволява да се изследват ситуации, които са трудноизпълними в практиката. Въпреки това акцентът е по-добре да падне върху динамичната верификация, която има по-добро практическо приложение в процеса на разработка и ясно структуриран подход при извършването на тестовете.

Провеждането на тестове може да протече по много различни начини, в зависимост от избраните методи за работа. Въпреки това практиката показва, че наличието на unit, integration, system и acceptance тестове в тази последователност дава най-добри резултати и завършеност на процеса *[виж Приложение 1]*. Изключването на която и да е от фазите в този процес би имало по-скоро негативни последици, тъй като различните видове тестове взаимно се допълват и зависят един от друг.

По-важен е въпросът и изборът на подходящ подход за всяка една от тези стъпки. Двата основни метода са black-box и white-box. „

White-box testing-а акцентира върху изходния код на приложението, което го прави особено подходящ за използване при unit тестовете. Това ще позволи по-лесното откриване и поправяне на възникналите грешки в съвсем ранен етап от разработката. Тестовете могат да бъдат писани от самите разработчици на софтуера, което е предпоставка за по-добро изследване на кода и дава възможност за самоанализ на програмиста.

Integration тестовете се прилагат върху вече завършени и интегрирани заедно звена от кода. Тяхната цел е най-вече откриването на проблеми в комуникацията между отделните модули, което най-често се дължи на пропуски в логиката на приложението. За това и поради вече доста по-големия обем на кода, изследван от integration тестовете, black-box методът е по-удачният избор. Абстрахирането от кода позволява на разработчиците на тестове да се фокусират[10]

10

върху функционалността и поведението на програмата. Има няколко подхода,[11] които се използват при intergation тестовете. Като най-удачен избор изглежда подходът отдолу-нагоре, при който акцентът пада върху най-ниското ниво. Малък недостатък е, че е необходимо всички или повечето модули на едно и също ниво на развитие да са готови, когато се провеждат тестовете, но при добра организация на работния процес това не би следвало да е проблем. За сметка на това, подходът предлага доста по-лесно откриване на грешките.

След успешното преминаване на всички тестове идва ред на изпробването на цялата система. Интегрирането на всички модули и взаимната им работа върху даден хардуер могат да предизвикат неочаквани резултати или да доведат до ситуации, които тестовете на по-ранен етап не са били в състояние да проверят. За това провеждането на системни тестове не бива да се пренебрегва. За тях отново най-подходящ се оказва black-box методът. White-box testing-а на този етап от динамичната верификация би бил много лош избор, тъй като количеството код, от което се състои целия продукт обикновено прави невъзможни тестове от този тип.

Макар на пръв поглед regression тестовете да изглеждат излишни след тези три етапа, те са задължителни след промени по кода или реализацията на програмата. Повторното провеждане на тестове е единствената гаранция, че промените не водят със себе си нови проблеми. Само unit тестването на променената част от кода е крайно недостатъчно, защото промени в една област могат да засегнат и други части от програмата, за които не подозираме.

От разгледаните подходи и методи за софтуерна верификация можем да заключим, че изборът на един единствен подход за целият процес не е удачен. Изборът на подходяща комбинация, една възможност за която е разгледана по-горе, е важна стъпка, която трябва да се направи предварително, при планирането на целия процес на разработка, взимайки предвид спецификата на заданието и очакваните мащаби, в които ще се развие проектът. За различни по мащаб проекти и в зависимост от това какъв е приоритетът на сигурната и коректна работа на програмата, могат да се използват различни съчетания на способи за работа.

В дългосрочен план, необходимостта от наличието на четирите стъпки в динамичната верификация (unit, integration, system и acceptance testing) остава все така голяма. С напредването на работата по даден проект, важността на acceptance тестовете расте, тъй като те са гарант за правилното функциониране на разработените по-рано модули. Що се отнася до white-box метода, няма изгледи за масовото му използване в по-късните етапи на верификация, поради необходимостта от много на брой и високо квалифицирани програмисти, които да бъдат запознати в дълбочина с големи части от проекта. Това е икономически необосновано, при наличието на black-box метода, който предлага също добра гаранция за коректност и надеждност на тестовете. Макар и по-слабо застъпена, статичната верификация остава без алтернатива за математически обосновано доказване верността на алгоритъма, съставляващ програмата.

11

Библиография

Използвани източници:

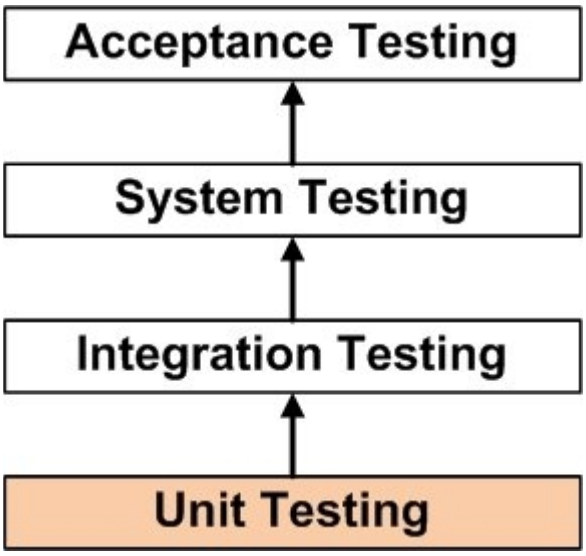
- http://en.wikipedia.org/wiki/Software_verification
- http://msdn.microsoft.com/en-us/library/windows/hardware/ff552806(v=vs.85).aspx
- http://www.cs.uakron.edu/~collard/cs680/notes/SEMDynamicVerification.html
- http://en.wikipedia.org/wiki/Black-box_testing
- http://en.wikipedia.org/wiki/White-box_testing
- http://www.guru99.com/black-box-testing.html
- http://www.guru99.com/white-box-testing.html
- http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx
- http://en.wikipedia.org/wiki/Unit_testing
- http://www.guru99.com/unit-testing.html
- http://en.wikipedia.org/wiki/Integration_testing
- http://msdn.microsoft.com/en-us/library/aa292128(v=vs.71).aspx
- http://www.guru99.com/integration-testing.html
- http://en.wikipedia.org/wiki/Regression_testing
- http://msdn.microsoft.com/en-us/library/aa292167(v=vs.71).aspx
- Building a Better Bug Trap, The Economist, June 2003
- http://www.codeproject.com/Articles/5579/Black-Box-Testing-Its-Advantages-and-Disadvantages
- http://creativetesters678.blogspot.com/2008/07/advantages-and-disadvantages-of-black.html[12]

12

Приложения

Приложение 1

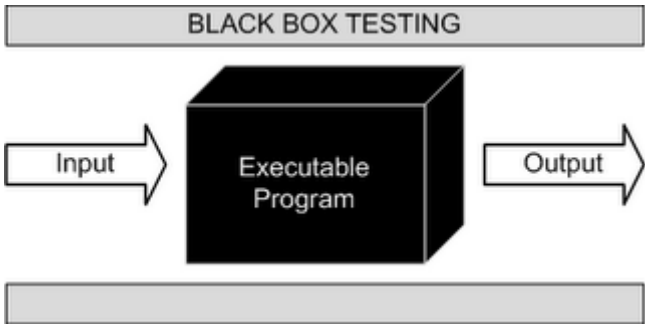
Фази на процеса на динамична верификация (тестване).



Източник: http://softwaretestingfundamentals.com/unit-testing/

Приложение 2

Илюстрация на идеята на black-box testing метода.

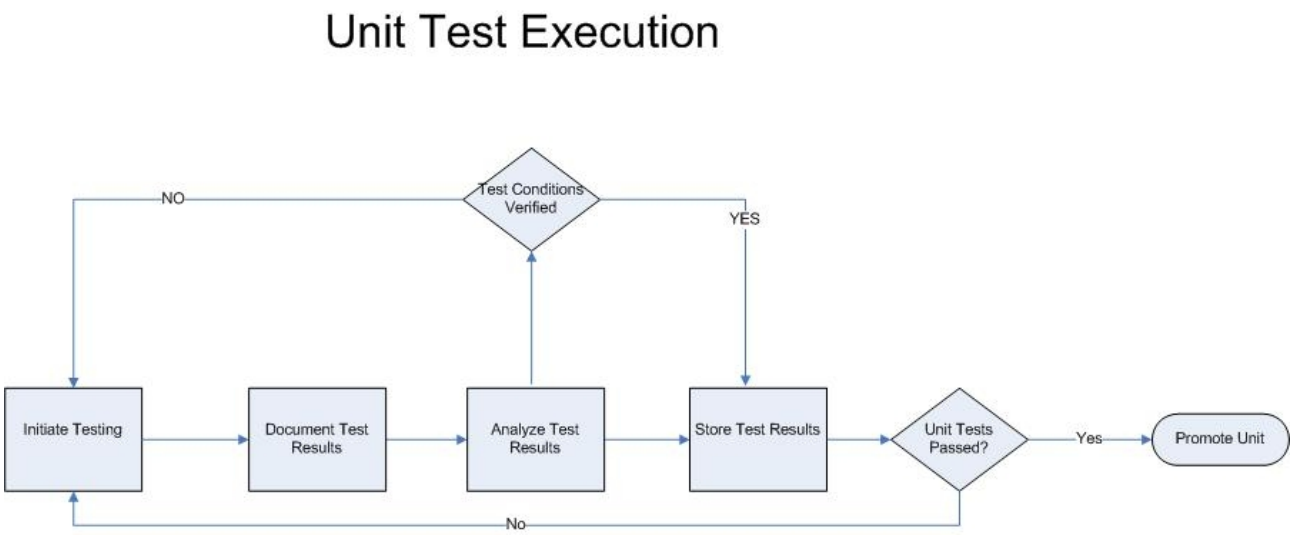


Източник: http://muzakstudyzone.blogspot.com/2012/12/white-and-black-box-testing-techniques.html

13

Приложение 3

Схема, описваща процеса на unit тестване.



Източник: http://efficientqa.com/Process/pm.html[14]

14