Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Anàlisi del test de software en aplicacions aeronàutiques-aeroespacials

TITULACIÓ: Enginyeria Tècnica Aeronàutica, especialitat Aeronavegació

AUTOR: Josep Bergay Ribas

DIRECTOR: José M. Yúfera Gómez

DATA: 11 de març de 2014

Títol: Anàlisi del test de software en aplicacions aeronàutiques-aeroespacials

Autor: Josep Bergay Ribas

Director: José M. Yúfera Gómez

Data: 11 de març de 2014

Resum

Els projectes aeronàutics i aeroespacials involucren una gran quantitat de treballadors i empreses diferents. Degut als alts nivells de seguretat que necessiten i la complexa interacció entre els diferents components, el software hi juga un paper clau.

Com s'assegura la qualitat d'aquest software i prèviament com es defineix la qualitat, és una part més del procés de desenvolupament. Molts professionals del sector tenen la percepció que haver de comprovar si el codi és correcte o compleix uns certs estàndards és una càrrega excessiva pel projecte i sense cap importància.

Tot i així, es demostra la seva importància quan molts dels accidents aeroespacials han tingut en la cadena de causes algun error de software.

Aquest document té com a objectiu aprofundir en com avaluar els requisits més tècnics associats a la qualitat d'un producte software amb finalitats aeroespacials.

Title: Software test analysis for aerospace applications

Author: Josep Bergay Ribas

Director: José M. Yúfera Gómez

Date: March, 10th 2014

Overview

Space and aeronautics are fields that involve an enormously big number of different people and companies. Due to the high levels of safety they require and because of the complex interaction between components, software plays a key role.

How the quality of this software is examined and how it is previously defined, it's just another part of the development process. Many professionals have the feeling that checking whether or not the code is correct or complies with certain standards is an undesirable and useless bulk load.

Nevertheless, it is shown their importance when quite a few of the aerospace accidents involved a software error somewhere in the error event chain.

The present document aims to deep into the most technical requirements linked with software quality and how they are evaluated. Keeping in mind, the constraints that aerospace missions bring.

ÍNDEX

ÍNDEX DE FIGURES

ÍNDEX DE TAULES

INTRODUCCIÓ

Les sondes de la NASA Mars Climate Orbiter i Mars Polar Lander, el vol inaugural de l'Ariane 5, el satèl·lit Milstar a bord d'un Titan IV/Centaur i la sonda SOHO, són segurament algunes de les missions aeroespacials més conegudes que han acabat fracassant degut a errors de l'equip tècnic o de tipus organitzatiu.

Però el que realment tenen en comú aquests accidents és que d'alguna manera el software hi va jugar un paper important. És més, quan les autoritats pertinents van investigar les causes, els informes (veure [1]) implicaven el software com a part de l'error, però mai es va determinar en quina part de tot el desenvolupament es va produir.

Una gran part de la indústria de software aeroespacial, degut als canvis de paradigma i en alguns casos probablement degut a un excés de confiança, ha donat per suposat que si alguna fase dels projectes s'ha de retallar ha de ser la part que es dedica a verificar el nivell més baix de component. Això demostra una falta de comprensió dels riscos derivats del software.

Aquest canvi de paradigma ja es va donar temps enrere quan als noranta el director de la NASA va pronunciar per primer cop el que seria la nova manera d'actuar de l'agència, el "Better, Faster, Cheaper". Aquestes paraules van sorgir en contraposició al "Higher, Faster, Farther" que havia estat manant fins aleshores. En aquella època, ja acabada la guerra freda, els programes espacials van patir un retallada important de recursos. Juntament amb el fet que molts dels productes aeroespacials ja havien estat desenvolupats, entraven a la seva vida útil i se n'esperava un rendiment. El que calia fer doncs, era una millora de la gestió global dels recursos, i els nous processos de desenvolupament havien de ser també més ràpids i per descomptat, més barats.

L'excés de confiança es pot explicar degut a les enormes modificacions que han anat patint tots els components involucrats en les missions espacials. Els enginyers han de decidir què es pot reutilitzar, què s'ha de modificar i què s'ha de desenvolupar de nou. I no sempre s'han realitzat tots els passos possibles per tal d'assegurar un correcte funcionament del component.

Aquest document té com a objectiu aprofundir en com avaluar els requisits més tècnics associats a la qualitat d'un producte software amb finalitats aeroespacials. No existeix una manera sistemàtica i generalitzada de realitzar-ho, així s'exploraran diferents estratègies i tècniques. Inclou la descripció d'un cas real de projecte de test, en el que he estat implicat, junt amb els seus informes i exemples, entre d'altres. També s'intenta donar una visió de quina és la importància del procés de proves dins del projecte software.

Al primer capítol es parla dels accidents més famosos en el sector aeroespacial que s'han esmentat amb anterioritat. També es parla dels sistemes de control i

es proposa una classificació dels diferents errors que es poden produir. Apareix la definició de qualitat i com es pot mesurar.

En el segon capítol s'aprofundeix en l'aspecte més tècnic i s'analitzen els mètodes de test segons diferents classificacions. La primera en nivells segons l'etapa de desenvolupament del projecte. La segona de tipus segons la seva finalitat. I la última segons com es dissenya i com s'executa la prova.

El tercer capítol conté informació sobre un projecte de test de software aeroespacial, vist des de la meva òptica d'enginyer de test o provador. He participat en la majoria de fases del procés de proves com poden ser el disseny o la implementació dels tests.

Concretament apareix l'estratègia emprada en les proves dinàmiques i estàtiques, amb alguns exemples. Només es mostren pocs exemples perquè s'ha de preservar la confidencialitat del projecte que demana el client.

L'últim capítol està reservat per a les conclusions sobre l'anàlisi del software en general. Finalment, s'explica la importància de testar el software i es reflexiona sobre l'impacte d'aquest al desenvolupament de la resta del projecte.

El document conté al final un glossari definint alguns conceptes o sigles i traduint altres termes en anglès que es fan servir al llarg del treball.

CAPÍTOL 1. NECESSITAT DEL TEST DE SOFTWARE

Tal i com s'ha esmentat en la introducció hi ha hagut més d'un accident aeroespacial on el software ha tingut un paper clau, ja sigui com a causa principal o més aviat secundària. El software ens permet construir sistemes amb un nivell de complexitat i acoblament que està més enllà de la nostra capacitat de control. De fet, aquests sistemes tenen tantes interaccions internes, normalment controlades per software, que no és possible planificar o entendre tots els casos.

El cas de la Mars Climate Orbiter(MCO) és un dels més coneguts i un dels més comentats ja que l'error va avergonyir a les agències i empreses implicades. La MCO era una sonda de la NASA amb finalitats científiques que havia d'orbitar el planeta roig. Entre alguns dels objectius es trobaven monitoritzar les condicions atmosfèriques, determinar la distribució d'aigua i registrar canvis a la superfície marciana. També havia de servir d'enllaç de comunicacions per a les següents missions al mateix planeta.

La sonda havia d'apropar-se a Mart fent servir la tècnica aerobraking per tal de poder posar-se en la posició correcte per entrar a l'òrbita adequada. Ja durant el trajecte interplanetari els tres mètodes que hi havia per calcular l'òrbita discrepaven entre ells, aquesta situació es va anotar però només es va notificar informalment i no va ser mai resolta. Quan es va procedir a calcular l'última trajectòria les estimacions deien que el periapsis amb el planeta havia de ser d'uns 226km ara bé els càlculs finals van ser en el rang de 150-170km, quan el mínim que podia sobreviure es trobava als 80km. La sonda va entrar en la zona sense senyal, darrere el planeta, quaranta-nou segons abans del previst. I ja no es va poder tornar a establir contacte amb la sonda.

La NASA va crear una comissió d'investigació (veure [1]) per determinar les causes de la pèrdua de senyal. El software recopilava les dades de propulsió del coet que més tard es feien servir per calcular trajectòries, d'aquests sistemes n'hi havien d'embarcats i del segment de terra. L'arxiu havia de contenir, per requisit, les dades de l'empenta en el sistema internacional (N-s). Doncs bé, el requisit no es va seguir i es va programar de manera que es guardessin els valors en sistema imperial (lbf-s). Així doncs, l'efecte de la trajectòria es va veure modificat en un factor 4.45 que és precisament el factor de conversió entre Newtons i lliures. Un defecte en el software, aparentment senzill i fàcil de detectar, acabava amb una missió de més de sis-cents milions de dòlars.

L'altre cas, també mític ja en el món aeroespacial, és l'explosió del llançador Ariane 5 en el seu vol inaugural. El llançador va tenir un comportament nominal fins el segon trenta-sis des de la posta en marxa dels motors. L'autodestrucció va ser desencadenada correctament. D'aquesta manera traçant inversament la cadena d'esdeveniments es va poder concloure que l'error va ser causat per una pèrdua de la informació d'actitud i guiatge trenta segons després del llançament.

El control de vol de l'Ariane 5 és bastant comú. Disposa de dos sistemes de referència inercials (SRI), amb software intern, que mesuren velocitats i angles i a través d'un bus de dades s'envien a l'ordinador de bord. L'ordinador de bord executa el pla de vol en funció dels paràmetres rebuts enviant la senyal als actuadors perquè controlin la propulsió. El disseny dels SRI era pràcticament el mateix que es feia servir en els Ariane 4, particularment en el que fa referència al software.

La raó per la qual hi va haver aquesta pèrdua d'informació tal i com va demostrar la investigació posterior és que els SRI van enviar dades que l'ordinador no sabia interpretar. El software intern dels SRI va llançar una excepció, que no era controlada, al intentar convertir una coma flotant de 64bits en un enter de 16bits. La funció en qüestió és deia HorizontalBias, i el que portava a terme era l'alineació de la plataforma inercial. Aquesta funció s'executava durant cinquanta segons. En el cas dels Ariane 4, quan es produïa la crida era en una seqüència bastant anterior a l'encesa de motors. En canvi en els Ariane 5, la crida de la funció es feia just tres segons abans, de manera que l'execució continuava ben entrat el vol. Al estar el coet en marxa un dels SRI va detectar un valor molt més gran de l'esperat i va intentar passar el control a l'altre, que també tenia el mateix problema, per tant, els dos es van apagar. Un cop sense SRI l'ordinador a bord va resoldre que era un desviament de trajectòria i va ordenar la neutralització del llançador.

L'origen no va ser la pròpia excepció de la funció, sinó que era que els SRI es van apagar. El gran desencert va ser la pròpia cultura del programa Ariane al pensar que les excepcions de software només podien indicar un error aleatori de hardware, i que per tant podien ser resoltes apagant el sistema i passant a un altre sistema redundant. El proveïdor dels SRI va complir amb les especificacions. El que va haver-hi va ser un greu error de disseny al pensar que el software que funcionava bé per als Ariane 4, per similaritat, també ho havia de fer per l'Ariane 5.

Els accidents que involucren software són sovint accidents de sistema que provenen més aviat d'interaccions entre els components que per un error d'un component en concret. En aquests casos, el software va fer alguna cosa que estava malament. Això no vol dir que el software fallés en cap moment. De fet, en molts altres casos els components software i hardware van funcionar individualment bé, però va ser la combinació de comportaments que portà a un accident catastròfic.

El software crític, aquell que apareix en sistemes de control crítics, s'ha d'especificar al detall, això inclou especificar els casos excepcionals, i s'ha de prendre una estratègia que controli els errors del software.

1.1. Sistemes de control crítics

Els sistemes de control crítics són aquells que han de garantir un bon funcionament de l'entorn per a poder complir amb els requisits, és a dir, els requisits involucren l'entorn. Normalment són entorns verdaderament molt extrems on qualsevol error del sistema podria posar en perill vides humanes. Per tant, no solament trobem aquests tipus de sistemes dins el món aeronàutic o aeroespacial sinó que també apareixen en altres sectors.

Podem dividir els components que participarien en el control d'una operació nominal en tres grans grups: hardware, software i persona. Normalment serà un sistema que combina els diferents grups anteriors per tal d'assolir un nivell de seguretat major.

El detonant original perquè el sistema falli pot venir de causes molt diverses. Des d'un error tècnic on s'hi inclouria els casos en què fallaria algun dels grups, fins a errors de tipus organitzatius o de falta de comunicació.

1.1.1. Errors de hardware

Els errors de hardware són els més fàcils d'evitar. Segons la teoria d'errors, aquests poden ser, de tres tipus: sistemàtics, d'escala o accidentals. Els sistemàtics apareixen sempre, per tant, han de ser primerament detectats fent experiments i posteriorment calibrats o controlats amb hardware o software. Per reduir un error d'escala la única manera és escollint un sensor amb una sensibilitat més gran. I els errors accidentals són els més difícils de preveure ja que són deguts a canvis aleatoris a l'escenari. Tot i això, la solució rau, simplement, en saber la probabilitat requerida i calcular quants components són necessaris per complir-la. Així doncs, només és qüestió de redundància fins a arribar a un percentatge que sembli segur o compleixi amb els requisits.

1.1.2. Errors humans

Molts dels sistemes de control acaben incloent una interacció home-màquina per tal d'oferir un altre bucle de control. El raonament o cadena de causes que pot portar a una persona a cometre un error és d'una complexitat molt elevada. Els errors humans són estudiats per un conjunt de matèries tant diferents com la psicologia o les matemàtiques i poden ser causats per multitud de factors.

Alguns tipus de tècniques d'anàlisi d'errors humans són les tècniques probabilístiques i les tècniques cognitives. Les primeres ofereixen un estudi de la persona com una peça de hardware de manera que es pugui incorporar la probabilitat d'error a un arbre de probabilitats de tot el sistema. Les segones intenten modelitzar el comportament humà segons el nivell de control que té l'operari i les característiques del seu entorn. Així doncs, si hi ha persones dins el bucle de control sempre hi haurà present un error humà latent. El càlcul de la

probabilitat d'error ha tenir en compte el temps durant el qual l'error latent pot persistir sense ser detectat. Aquests mètodes serviran justament per analitzar la compenetració dels subsistemes o procediments amb la naturalesa humana, i posteriorment dissenyar plans d'entrenament nous o procediments nous.

1.1.3. **Errors de software**

El paper del software en els sistemes de control crítics ha anat augmentant considerablement amb els anys. Fins al punt, que avui en dia és inconcebible implementar un sistema d'aquestes característiques sense software. Per aquest motiu, la qualitat del software s'ha convertit en un factor determinant per assolir l'èxit de sistemes i missions.

Un error de software o bug és un comportament de software que és incorrecte o inesperat. Els orígens dels errors poden ser de diferents tipus. Tant poden ser defectes en el codi com defectes en el disseny del programa. També ho són si es sotmet el software a entorns - sistemes operatius, compiladors, dispositius de hardware - pels que no ha estat dissenyat. En aquest últim cas el que fallaria seria el software però no seria el verdader responsable, ho seria la persona o l'equip que ha pres la decisió.

És l'objectiu d'aquest treball explicar quins mètodes utilitzar per a aconseguir un software de control d'un sistema crític amb la màxima qualitat possible.

1.2. **Definició de la qualitat de software**

La nova sèrie 25000 de normes ISO/EC defineix la qualitat software com: la totalitat de la funcionalitat i prestacions d'un producte software que estan relacionades amb la seva capacitat de satisfer les necessitats explícites o implícites (veure [2]). Aquest model de qualitat està compost per les vuit característiques següents amb les subcaracterístiques corresponents "en negreta":

1.2.1. **Adequació funcional**

Representa la capacitat del producte software per a proporcionar funcions que satisfan les necessitats declarades i implícites, quan el producte es fa servir en les condicions especificades.

Completesa: grau en el qual el conjunt de funcionalitats cobreix totes les àrees i els objectius especificats per l'usuari o el client.

Correcció: capacitat del producte o sistema per a proveir resultats correctes amb el nivell de precisió requerit.

Adequació: capacitat del producte software per a proporcionar un conjunt apropiat de funcions per a tasques i objectius d'usuari especificats.

1.2.2. **Eficiència**

Aquesta característica representa l'exercici relatiu a la quantitat de recursos utilitzats sota unes condicions determinades.

Comportament temporal: els temps de resposta i processat i els ràtios de rendiment d'un sistema quan porta a terme les seves funcions sota condicions determinades en relació amb un banc de proves establert.

Utilització de recursos: les quantitats i tipus de recursos utilitzats quan el software porta a terme la seva funció sota condicions determinades.

1.2.3. **Compatibilitat**

Capacitat de dos o més sistemes o components per a intercanviar informació i/o dur a terme les seves funcions quan comparteixen el mateix entorn hardware o software.

Coexistència: capacitat del producte per a coexistir amb un altre software independent, en un entorn comú, compartint recursos comuns sense detriment.

Interoperabilitat: capacitat de dos o més sistemes o components per a intercanviar informació i utilitzar aquesta informació.

1.2.4. **Usabilitat**

Capacitat del producte software per a ser entès, après, usat i resultar atractiu per a l'usuari, quan es fa servir sota determinades condicions.

Capacitat per a reconèixer la seva adequació: capacitat del producte que permet a l'usuari entendre si el software és adequat per a les seves necessitats.

Capacitat d'aprenentatge tècnic: capacitat del producte que permet a l'usuari aprendre la seva aplicació.

Capacitat per a ser usat: capacitat del producte que permet a l'usuari operar-lo i controlar-lo amb facilitat.

Protecció contra errors d'usuari: capacitat del sistema per a protegir als usuaris de cometre errors.

Estètica de la interfície d'usuari: capacitat de la interfície d'usuari d'agradar i satisfer la interacció amb l'usuari.

Accessibilitat tècnica: capacitat del producte que permeti que sigui utilitzat per usuaris amb determinades discapacitats.

1.2.5. **Fiabilitat**

Capacitat d'un sistema o component per a exercir les funcions especificades, quan es fa servir sota unes condicions i un període de temps determinats.

Maduresa: capacitat del sistema per a satisfer les necessitats de fiabilitat en condicions normals.

Disponibilitat: capacitat del sistema o component d'estar operatiu i accessible per a el seu ús quan es requereixi.

Tolerància a errors: capacitat del sistema o component per a operar segons el previst en presència d'errors hardware o software.

Capacitat de recuperació: capacitat del producte software per a recuperar les dades directament afectades i restablir l'estat desitjat del sistema en cas de interrupció o fallada.

1.2.6. **Seguretat**

Capacitat de protecció de la informació i les dades de manera que persones o sistemes no autoritzats no puguin llegir-los o modificar-los.

Confidencialitat: capacitat de protecció contra l'accés de dades i informació no autoritzades, ja sigui accidental o deliberat.

Integritat: capacitat del sistema o component per a preveure accessos o modificacions no autoritzades a dades o programes d'ordinador.

No repudi: capacitat de demostrar les accions o els esdeveniments que han tingut lloc, de manera que dites accions o esdeveniments no puguin ser repudiats posteriorment.

Responsabilitat: capacitat de rastrejar de forma inequívoca les accions d'una entitat.

Autenticitat: capacitat de demostrar la identitat d'un subjecte o d'un recurs.

1.2.7. **Mantenibilitat**

Aquesta característica representa la capacitat del producte software per a ser modificat efectiva i eficientment, degut a necessitats evolutives, correctives o perfectives.

Modularitat: capacitat d'un sistema o programa (compost de components discrets) que permet que un canvi en un component tingui un impacte mínim als altres.

Reusabilitat: capacitat d'un actiu que permet que sigui utilitzat en més d'un sistema software o en la construcció d'altres actius.

Analitzabilitat: facilitat amb la que es pot avaluar l'impacte d'un determinat canvi sobre la resta del software, diagnosticar les deficiències o causes d'errors en el software, o identificar les parts a modificar.

Capacitat per a ser modificat: capacitat del producte que permet que sigui modificat de forma efectiva i eficient sense introduir defectes o degradar el funcionament.

Capacitat per a ser provat: facilitat amb la que es poden establir criteris de prova per a un sistema o component i amb la que es poden portar a terme les proves per a determinar si es compleixen aquests criteris.

1.2.8. **Portabilitat**

Capacitat del producte o component de ser transferit de forma efectiva i eficient d'un entorn hardware, software, operacional o d'utilització a un altre.

Adaptabilitat: capacitat del producte que li permet ser adaptat de forma efectiva i eficient a diferents entorns determinats de hardware, software, operacionals o d'ús.

Capacitat per a ser instal·lat: facilitat amb la que el producte es pot instal·lar i/o desinstal·lar amb èxit en un determinat entorn.

Capacitat per a ser reemplaçat: capacitat del producte per a ser utilitzat en lloc d'un altre producte software determinat amb el mateix propòsit i en el mateix entorn.

CAPÍTOL 2. MESURA DE LA QUALITAT SOFTWARE

Observant les característiques que composen la definició de la qualitat software, veiem que són atributs molt diferents i que hauran de tenir procediments de proves diferents per a poder avaluar-les. En la majoria dels casos cada característica tindrà el seu conjunt de proves, tant poden ser quantitatives, qualitatives o una barreja de les dues.

L'ideal seria un software amb el màxim nivell de qualitat. Però, també s'han d'establir prioritats perquè hi ha característiques de la qualitat que poden xocar entre elles com ara la seguretat i la mantenibilitat o l'eficiència i la portabilitat.

Cal distingir el que s'anomenen activitats constructives de les analítiques. En les primeres, el que es vol assegurar és que tot el procés de desenvolupament del software sigui el més lliure de defectes possibles i per tant, prevenir errors. Es dota a l'equip amb guies, estàndards, requisits legals, eines, entorns de desenvolupament integrats i tots els mètodes d'enginyeria de software tant organitzatius com més tècnics.

Per contra, la missió de les activitats analítiques és detectar els errors i corregir-los. Consisteixen en proves sobre el codi ja desenvolupat. En la majoria dels casos, degut a la gran complexitat dels softwares aeroespacials, no és possible realitzar proves exhaustives. S'ha d'escollir els casos més interessants segons el criteri de qualitat més prioritari. Inclús, si es pogués testar tot al cent per cent no tindríem la certesa de que el codi està lliure de defectes. Això sí, tindrem un grau més alt de confiança que si no es porten a terme aquestes proves.

A continuació, s'explica el conjunt de proves analítiques de software. Assumim un model de projecte en V, tal i com mostra la Fig. 2.1, ja que és el més comú i il·lustra molt bé la relació entre provadors i desenvolupadors.

2.1. Nivells

A cada activitat de la branca de desenvolupament l'hi correspon un nivell de test de la branca de proves. Es pot fer l'analogia dels nivells de test amb els nivells de requisits. Els requisits a nivell de sistema dissenyarien i validarien els tests de sistema i els requisits de baix nivell es comprovarien mitjançant els unitaris o d'integració.

2.1.1. Test de component

Els tests de component són els de nivell més baix dins un sistema. Cada llenguatge defineix el seu component d'una manera en concret (mòdul en c,

classe en JAVA o C++, etc.). A aquest nivell de test també se'n diu test unitari, unit test en anglès, o test de baix nivell.

És recomanable que aquestes proves les dissenyin els mateixos desenvolupadors ja que ho poden fer servir com a comprovant que aquella funció o classe funciona correctament i són qui coneix millor el codi.

Si es vol un alt grau de confiança, cal provar tots els components un per un. Ara bé, si per pressupost, temps o qualsevol altra raó no fos possible s'haurien d'escollir els casos de prova. Un criteri pot ser escollir components amb funcionalitats semblants i fer tant sols un test. Un altre seria fer els tests de component dels subsistemes més crítics.

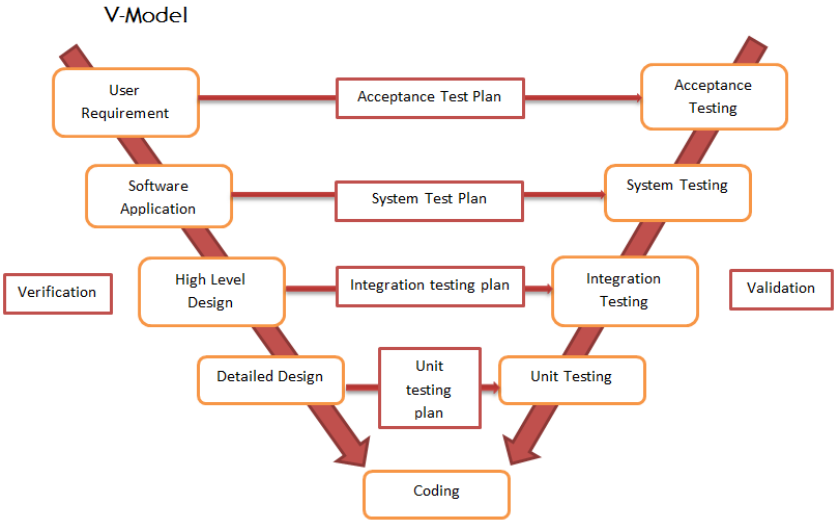


Fig. 2.1 Diagrama representant el model V de desenvolupament d'un projecte software

En el cas que el component tingui dependències i no pugui ser provat tot sol, es recorrerà a l'ús de stubs i drivers per simular el comportament d'allò que no està sota el nostre focus del test.

Cal tenir present que s'han d'efectuar tant proves positives o nominals com proves negatives o d'error. És a dir, per a cada test de component s'haurà d'haver generat prèviament un conjunt de dades de entrada tant vàlides com no. Molts cops és interessant justament el que fa la funció en casos d'error.

Normalment es requereixen eines externes al compilador per a poder executar els tests. El tipus de marc de test unitari més utilitzat és XUnit, un exemple del qual seria JUnit per JAVA, veure [3]. És possible però, realitzar tests unitaris sense cap eina, dotant el codi de mecanismes de funcions de test que afirmin que la sortida del component és l'esperada, llibreries especials o mecanismes de gestió d'excepcions, etc.

2.1.2. **Test d'integració**

Un nivell per sobre les de component estan les proves d'integració. El seu objectiu és confirmar que la comunicació entre els diferents components es porta a terme adequadament. És a dir, que no hi podem detectar defectes encara que la sotmetem a uns entorns de prova difícilment reproduïbles en un entorn real. Tant es pot integrar components com subsistemes.

Al igual que amb les proves de component, si s'ha de simular altres subsistemes o generar paràmetres d'entrada es pot recórrer a l'ús d'stubs i drivers.

Per a dissenyar aquest nivell de test es requereix coneixement de l'arquitectura del software. Es volen proves el més ajustades possible. Hi ha vàries estratègies a seguir. En cada projecte s'escollirà l'estratègia en funció del cost, en funció del temps de finalització del desenvolupament dels components i en funció del grau d'eficiència de les proves, entre d'altres. A continuació, s'expliquen les estratègies a seguir més comunes.

2.1.2.1 Ascendent ("top-down")

Aquest és un enfocament d'integració on es comença provant els components de més alt nivell. Si el resultat és correcte, es procedirà a seleccionar un dels mòduls i a analitzar-lo i així consecutivament.

És una estratègia que requereix que tot el programa estigui desenvolupat o un ús considerable d'stubs. Això pot suposar un problema, si el que hem de simular és justament un dels mòduls afectats per la comunicació que volem provar. Tampoc és factible realitzar aquesta aproximació fins que no hi hagi bastants components ja desenvolupats.

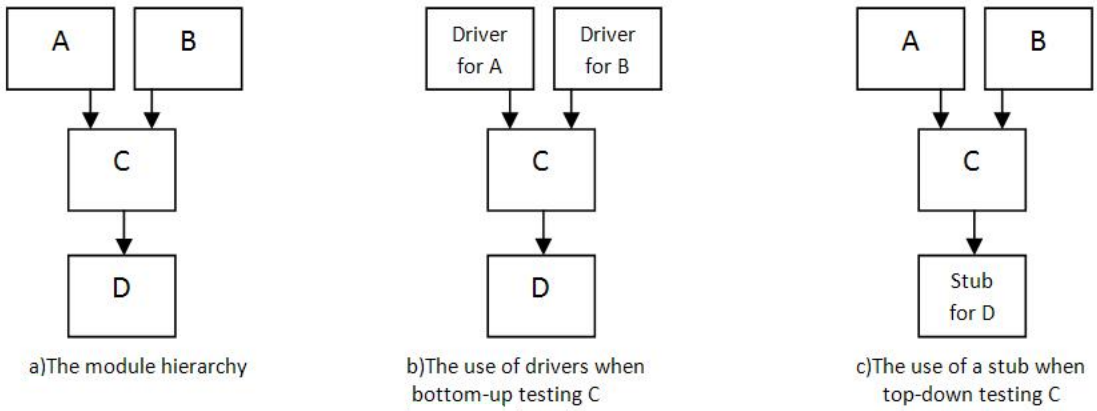


Fig. 2.2 Exemple d'ús de stubs i drivers per fer test d'integració amb el mòdul C, veure [4]

Per contra, si s'ha de recórrer als stubs, un dels avantatges és que t'assegures un disseny correcte des del principi.

2.1.2.2. Descendent ("bottom-up")

L'estratègia d'integració descendent consisteix en realitzar els test de comunicació entre els components de nivell més baix. El següent pas és provar els mòduls superiors, fins a arribar al mòdul a dalt de tot de l'estructura del software.

Aquesta aproximació afavoreix el desenvolupament, ja que si el test falla és molt fàcil detectar l'error i corregir-lo. Si el projecte software consisteix en ampliar i/o millorar l'aplicació, aquesta és l'estratègia a seguir ja que l'estructura ja ve donada.

El desavantatge més evident és que es pot donar el cas que els tests s'hagin de refer si hi ha un error de disseny no detectat.

2.1.2.3 Big Bang

Aquesta és la solució més ràpida però més perillosa a la vegada. S'integren tots els components del sistema de cop. Aleshores es dissenyen els tests d'integració. S'escullen però només els més interessants segons si és un mòdul crític o segons altres criteris, ja que al integrar-ho tot hi ha masses interaccions a comprovar.

No és l'estratègia més adequada pels tests d'integració d'aplicacions software per a sistemes crítics. És una mica optimista pensar que la major part del programa és correcte. I en el cas de detectar un error requereix molt més temps que altres mètodes detectar on està situat.

Caldrà tenir en compte el compromís entre cost, confiança i temps de l'estratègia d'integració per a cada projecte a l'hora d'escollir la millor opció. Normalment s'escull una barreja entre ascendent i descendent, per tenir tant la seguretat d'una com la eficiència de l'altre.

2.1.3. **Test de sistema**

Les proves a nivell de sistema són les primeres proves que necessiten que tot el software estigui desenvolupat. Les proves es dissenyen des de l'òptica de l'usuari final. Cada test consisteix en una sèrie d'accions que el provador ha de realitzar amb l'aplicació en funcionament. Aquestes accions són els requisits

pactats amb el client, normalment requisits d'alt nivell o de sistema, o també els casos d'ús.

Òbviament en els casos de software crític no és factible posar el sistema en funcionament mentre encara faltin proves per fer. Primer per seguretat, no tindria sentit totes les precaucions preses abans, en el cas d'error es podria danyar l'entorn on opera el sistema. I darrerament per cost, ningú contempla que per exemple, un test de sistema del software de seguiment d'un coet necessiti d'un llançament real només per a verificar la prova.

En molts d'aquests casos la solució és crear una plataforma el més idèntica possible a la plataforma final. I aleshores, amb ajuda d'eines externes es simula l'entorn, les dades i tot el necessari, per arribar a una execució del sistema el més real possible.

És en aquest nivell on es pot avaluar també per primer cop les propietats esmentades al primer capítol que componen la qualitat software. Arribar a acordar una manera de quantificar les propietats no funcionals és molt difícil d'aconseguir. Així els requisits ja s'escriuen d'una forma bastant vaga. Cada persona tindrà la seva opinió respecte si una cosa és prou eficient, prou agradable visualment o prou intuïtiva. Tot i això, el client espera que siguin complerts.

2.1.4. **Test d'acceptació**

Els test d'acceptació són molt similars als de sistema en el sentit que s'avalua l'aplicació de manera global. La gran diferència però radica en qui les duu a terme. El propi client és qui dissenya les proves i les executa. Es pot donar el cas que el client tingui coneixements de software i pugui estar involucrat en altres nivells de test, però no és el més habitual.

El client procedirà a comprovar que tot un bloc de requisits es compleixen. Els requisits apareixen en el contracte, per tant, la resolució dels test d'acceptació determinarà el compliment de contracte. Aquestes proves es fan servir molts cops també com a fites dins el contracte, a l'hora de cobrar, en la planificació, etc. Es pot dir que quan un software ha passat les proves d'acceptació, aquella ja és una versió estable i llesta per al client.

2.2. **Tipus**

El que s'ha explicat fins ara són els nivells de test. Ara bé, cada nivell està pensat amb un objectiu diferent en ment. Així doncs, no podem fer servir el mateix estil de prova en tots els nivells.

2.2.1. **Funcionals**

Les proves funcionals són les que més es troben ja que poden aparèixer en cada nivell de test. L'altre raó és perquè de les propietats de la qualitat software comproven majoritàriament la més important, l'adequació funcional. L'altra propietat que avaluen encara que en menor mesura és la seguretat.

Aquestes proves són les típiques proves d'entrada i sortida. Cada prova té el seu criteri de sortida i uns possibles valors d'entrada. Si el criteri es compleix la prova es finalitza. D'altre manera, s'ha trobat un error i cal corregir i tornar a passar el test.

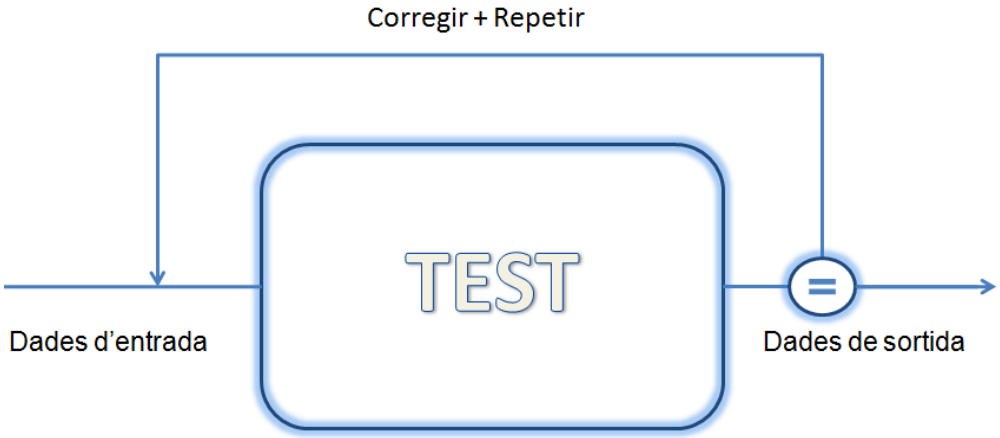


Fig. 2.3 Diagrama del funcionament d'un test funcional

També s'ha de tenir en compte que potser el codi conté algun error però la prova o el criteri no són els més adequats i és incapaç de detectar l'error. O si els valors d'entrada són dinàmics, és a dir, que varien en funció d'alguna cosa com ara del test executat anteriorment. En tots aquests casos el test no compliria la seva feina, avisar si hi ha errors.

Realitzar test funcionals a tots els nivells, tant nominals com d'error i per tots els paràmetres d'entrada possibles no és viable. És feina de l'equip de disseny de proves junt amb la gestió del projecte escollir el joc de proves més adequat per a cada projecte.

2.2.2. **No funcionals**

Els tests no funcionals examinen més aviat de quina manera realitza les tasques el software. Com s'ha comentat abans és complicat quantificar aquestes característiques. Si els requisits són molt indefinits és qüestió d'anar iterant amb el client fins que les dues parts quedin satisfetes.

L'altre manera és mirar d'avaluar aquests components no funcionals de la qualitat mitjançant un o més paràmetres funcionals. I arribar igualment a un acord amb el client que determinarà amb quin grau el requisit queda satisfet. Per exemple, es pot decidir que un software és més mantenible que un altre si el codi té més línees de comentaris.

Encara que les proves no funcionals són més pròpies dels nivells de sistema o acceptació, també es poden realitzar a tots els altres nivells. Algunes de les proves que s'inclourien en aquest grup són les proves de usabilitat, les proves de mantenibilitat.

2.2.3. **Estructurals**

Les proves estructurals o de cobertura consisteixen en comprovar que realment el software compleix els diagrames de flux dissenyats inicialment, que no hi ha codi mort, etc.

Normalment s'aconsegueix aquest tipus d'anàlisi mitjançant eines externes, les quals recopilen quines crides es fan, quines branques es segueixen, i van calculant el tant per cent de línees executes sobre el total. El resultat pot ser la cobertura de línees de codi, decisions, voltes a bucles, entrades a funcions, entrades i sortides a funcions, etc.

És recomanable executar-los junts amb els funcionals, així s'aconsegueix estalviar feina al provador i també s'obté un conjunt de resultats bastant coherent.

2.2.4. **Regressió**

L'objectiu dels tests de regressió és que si s'han produït canvis en el codi s'ha de verificar que tot continua sent correcte o que les proves funcionals continuen donant els mateixos resultats. És molt possible que a l'afegir una funcionalitat nova, el codi es vegi afectat més enllà de la pròpia funció o mòdul. Així totes les dependències hauran de ser també avaluades.

També acostuma a passar que els tests no siguin executables si ha canviat ja sigui el disseny o una variable nova. En aquest cas s'han de tornar a reprogramar.

Si el projecte de test està automatitzat resulta molt fàcil tornar a passar els test per confirmar que les funcionalitats abans provades segueixen igual. A continuació el que cal fer és analitzar la part modificada per dissenyar nous casos de test i incloure'ls al projecte.

2.3. Tècniques

Hem classificat les proves per nivells segons qui era el nostre objecte de test, també les hem classificat segons el tipus d'avaluació que feien si era més quantitativa o no. És possible classificar els tests segons com es duen a terme o segons quines eines es fan servir.

2.3.1. Estàtiques

Degut al volum de paràmetres que s'han de controlar i al volum de tests funcionals possibles, es va desenvolupar una manera de intuir la qualitat del software sense haver d'executar-lo.

L'anàlisi estàtic consisteix en verificar que el codi compleix uns estàndards de programació, que compleix el disseny, o que compleix alguns requisits. Existeixen aplicacions dissenyades especialment per avaluar el codi estàticament. Cosa que resulta molt més ràpida i menys costosa, que haver de verificar-lo manualment.

Avui en dia, els mateixos compiladors ja efectuen gran part d'aquest anàlisi, però és conjuntament amb els analitzadors que es pot fer un anàlisi estàtic total. Alguns dels paràmetres més analitzats són el flux de control, mitjançant grafs es poden detectar defectes com branques mortes. L'altra cosa que es pot mirar sense haver d'executar el programa és el flux de dades o d'estats. Una variable pot ser representada com una seqüència d'estats. Si alguna d'aquestes seqüències conté una subseqüència que no tingui sentit, aleshores el que hi ha és una anomalia en el flux de dades. Per exemple, no es pot referenciar una variable si està indefinida.

En general, les mètriques que apliquen aquestes eines tenen aspectes molt diferents. Així, mitjançant normes aplicables solament al codi, es pot garantir que el programa està des de ben comentat, fins a que no té pèrdues de memòria, passant per la complexitat ciclomàtica, entre d'altres.

2.3.2. Dinàmiques

Les tècniques dinàmiques a diferència de les estàtiques si que requereixen una execució del programa per a poder avaluar-lo. Cada cas de prova ha de tenir els següents components ben definits perquè el test tingui sentit: valors d'entrada, precondicions, resultats esperats, postcondicions, identificador i requisits. Els casos de prova poden ser creats a partir de requisits, aleshores la traçabilitat és important mantenir-la sempre per saber quin test verifica quin requisit.

Es divideixen en dos grans grups. La divisió es produeix en base al que coneix el provador sobre l'objecte i en base al que volem verificar.

2.3.2.1. Caixa blanca

Imaginant el nostre objecte de proves com si fos una caixa, aquest mètode és diu caixa blanca precisament perquè podem veure el que hi ha dintre. L'analogia s'aplica al software si coneixem l'estructura interna del component o subsistema a testar. És a dir, si tant a l'hora de dissenyar el test com a l'hora d'executar-lo, el provador pot disposar en tot moment del codi, de la documentació referent a diagrames de flux de control, components, etc.

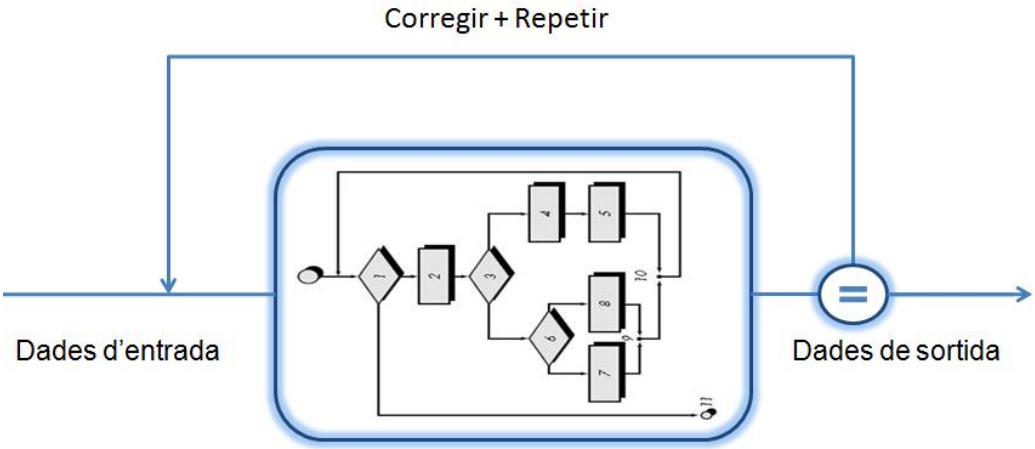


Fig. 2.4 Conceptual de caixa blanca amb diagrama de flux

Veurem que l'altre mètode no ens permet tenir aquesta informació, per tant, l'objectiu d'aquestes proves és verificar l'estructura interna de l'objecte. Per a realitzar aquests tests es necessiten eines que monitoritzin l'execució del codi. Aquests programes auxiliars avui en dia ja hi ha entorns de desenvolupament en els que estan integrats. Són necessaris perquè el que mesuren són paràmetres com la cobertura del codi, els perfils en l'ús de memòria, detecten colls d'ampolla, etc. Si es fes manualment seria molt costós pel propi provador.

El més mesurat és sense dubte la cobertura del codi. La cobertura del codi es dóna en percentatge i representa quantes de les opcions s'han executat sobre el total de la prova. Al mesurar simplement quin percentatge d'estructura s'executa no necessitem que el test sigui funcional. Així en les proves de cobertura és molt habitual l'ús d'stubs per tot allò extern a l'objecte.

No poder arribar a una cobertura d'un cent per cent implica que hi ha codi mort dins el programa. Codi mort que com que mai s'executarà és susceptible a induir falles de seguretat. I també pot implicar greus errors de disseny.

Hi ha diferents nivells de cobertura. Encara que la classificació no està generalitzada, el nivell superior és el de cobertura de funció. Els analitzadors de cobertura el que fan és observar quantes vegades s'ha cridat una funció i comprovar que la línia d'execució faci el retorn de la funció.

El següent és el nivell de cobertura de bloc. S'avalua quins segments de codi limitats normalment per "{ }" s'executen; cada funció té com a mínim un bloc. Els blocs es poden categoritzar en sentències, decisions i bucles. Les sentències són els blocs que estan explícits en el codi. Les decisions són els blocs implícits. És a dir, per a cada resultat de la condició hi ha dues possibles decisions com a mínim mentre que no té perquè haver-hi dues sentències. El cas més típic és un IF sense el seu ELSE. La cobertura de bucle consisteix en aconseguir executar el bucle, tipus FOR o WHILE, de diferents maneres. En funció de la nostra eina d'anàlisi podrem fer més o menys iteracions. El més comú és analitzar tres comportaments: que el bloc de dins el bucle s'executi dos o més cops, un cop i cap cop. Aconseguir un cent per cent de cobertura de bucles implica un cent per cent de cobertura de decisions i per tant també de sentències. La diferència és que es necessiten més casos de test per bucles que per sentències.

Segons la cobertura a nivell de condició, per a cada bloc no es mira la sortida de la condició sinó que s'avaluen casos segons la complexitat de la condició. Si és una condició formada a la vegada de condicions més petites mitjançant operadors lògics es pot extreure la cobertura de les possibles entrades a la condició. Hi ha tres tipus de cobertura de condició: condició simple, condició múltiple i mínim de condició múltiple.

En la condició simple només s'ha d'imposar que cada subcondició prengui com a mínim un cop els valors vertader i fals. Pot passar que el resultat de la condició global no canviï.

Taula 2.1. Exemple de casos de test de cobertura de condició simple

Condició: (i > 0 && b == true)		
i	b	result
i = 0 (false)	b = true (true)	false
i = 1 (true)	b = false (false)	false

Si el que es vol és fer un anàlisi de cobertura de condició múltiple s'ha de testar totes les combinacions possibles entre les subcondicions. Això pot comportar una explosió dels casos de prova, ja que augmenta exponencialment amb el

número de subcondicions presents. La condició múltiple garanteix cobertura de sentència i decisió.

Taula 2.2. Exemple de casos de test de cobertura de condició múltiple

Condició: (i > 0 && b == true)		
i	b	result
i = 1 (true)	b = true (true)	true
i = 1 (true)	b = false (false)	false
i = 0 (false)	b = true (true)	false
i = 0 (false)	b = false (false)	false

Finalment, la mínima cobertura de condició múltiple és molt semblant a la condició múltiple pura. Es busquen tots les combinacions possibles entres les subcondicions, com a l'anàlisi anterior, però aquest cop es descarten les que canviant alguna de les subcondicions el resultat no canvia. Els principals avantatges són que no necessita tants casos com l'altre, té en compte la complexitat de la condició i també garanteix la cobertura de sentència i decisió.

Taula 2.3. Exemple de casos de test de mínima cobertura de condició múltiple

Condició: (i > 0 && b == true)		
i	b	result
i = 1 (true)	b = true (true)	true
i = 1 (true)	b = false (false)	false
i = 0 (false)	b = true (true)	false
~~i = 0 (false)~~	~~b = false (false)~~	~~false~~

Es pot afirmar que en funció de la profunditat de cobertura desitjada el procés de proves pot arribar a ser molt costós. Arribar a un cent per cent de cobertura de bucles és possible en aplicacions molt senzilles, però en programes complexos s'ha d'arribar a un acord amb el client per reduir el número de casos de test i per tant reduir la càrrega de treball de l'equip de proves.

Hi ha altres criteris a l'hora de realitzar els test de cobertura com el JJ-path [5] (jump to jump path) o el d'estats, si és una màquina d'estats finits. Són tots molt semblants, hi ha dependències entre tots ells així el criteri a seguir s'ha d'escollir i mantenir al llarg de tot el projecte per l'equip de test.

Així doncs, l'objectiu de les proves de cobertura és verificar que el disseny i la implementació de l'estructura s'han portat a terme correctament. És habitual en els sistemes de control crítics que hi hagi algun requisit que demani un cent per

cent d'algun tipus de cobertura. El que permet l'anàlisi de cobertura és veure que les àrees que rarament són executades existeixen, i dota a l'equip de test d'una mesura més, entre d'altres, per verificar que no hi hagi regressions al llarg del desenvolupament del projecte. Això si, aquest mètode no pot detectar parts no implementades o requisits que falten.

Tot i que es pot realitzar anàlisi de cobertura a qualsevol nivell tant de component, d'integració o de sistema, el més comú és utilitzar-los a nivell de component ja que dissenyar un test de cobertura a un nivell major pot resultar molt complicat.

2.3.2.2. Caixa negra

La idea principal darrere aquesta tècnica és que el provador considera l'objecte de proves com una caixa negra. És a dir, ell no sap què hi passa dins de la funció o mòdul, l'únic que pot fer és posar-li uns valors d'entrada, executar i mirar a veure quin són els paràmetres de sortida.

Com que en principi, no podem mesurar cobertura perquè no coneixem l'estructura interna, es posa el focus sobre la funcionalitat del component, es verifiquen la correcció i la completesa d'una funció. Així doncs, els casos de prova estan basats en les especificacions. Solament mirant la capçalera i els comentaris, si estan ben fets, o amb la documentació s'hauria de poder decidir quins són els valors d'entrada i definir els criteris de sortida. En cas contrari el provador haurà de recórrer a l'experiència i/o a la intuïció.

Hi ha diferents mètodes de caixa negra. La partició en classes d'equivalència és el primer. Aquest mètode té dues aproximacions: treballar amb els valors d'entrada i observar el comportament a la sortida, o treballar amb els valors de sortida i obtenir els valors d'entrada. L'última aproximació requereix molt d'esforç, ja que les dades d'entrada han de ser obtingudes recursivament. Es fa servir doncs la primera.

Consisteix en agrupar els valors d'entrada en funció de la resposta que tindrà el programa. Tots els valors dins d'una classe d'equivalència hauran de ser entrades vàlides o no vàlides. Així per una mateixa prova podrem tenir diferents classes vàlides i d'error. I agafarem un sol de tots els valors dins d'una classe per cada test. Per tant, en un primer moment hi haurà tants tests com classes d'equivalència. A continuació un exemple senzill il·lustra la divisió en classes d'equivalència. Tenim una funció que te dos valors numèrics com a paràmetres d'entrada. Un representa un tant per cent i l'altre un valor positiu.

Taula 2.4. Exemple mètode divisió en classes d'equivalència

Variable	Classe	Estat	Valor	T1	T2	T3	T4	T5	T6
Número positiu	n > 0	Vàlid	1	o			o	o	o
	n =<0	No Vàlid	-1		o				
	n = valor no numèric	No Vàlid	"x"			o			
Percentatge	0%=<n=<100%	Vàlid	50%	o	o	o			
	n < 0 %	No Vàlid	-50%				o		
	n > 100 %	No Vàlid	150%					o	
	n = valor no numèric	No Vàlid	"x"						o

Tal i com veiem a la taula anterior el primer test (T1) és l'únic que hauria de tindre sortida nominal, i tots els altres cinc tests comptarien com a casos d'error. A priori no volem executar tests amb més d'una variable en estat invàlid ja que l'error podria quedar emmascarat.

Els valors escollits en cada test per representar la classe d'equivalència s'escullen també segons diferents ordres. Poden ser els més freqüents, els més sospitosos de produir algun error o simplement els valors límit. Els valors límit és el mateix límit exacte i els següents valors possibles tant vàlids com no. En el cas anterior suposant que es tractessin d'enters, els valors límits pel número positiu serien el -1; 0 i 1. I els valors límits pel percentatge serien -1; 0; 1 i 99; 100 i 101.

El mètode de la divisió en classes d'equivalència és molt eficaç a l'hora de dissenyar els diferents casos de prova. També té l'avantatge que sistemàticament apareixen els tests positius i els negatius. El principal inconvenient és que no tenen en compte els efectes de les dependències i combinacions i resulten menys eficaços en casos no numèrics. L'altre desavantatge és que les funcions molt complexes comportaran un gran número de casos de test.

L'altre mètode de caixa negra és el de transició d'estats. Aquest mètode a diferència de l'anterior té en compte les accions realitzades pel programa en el passat. Mitjançant diagrames de transició d'estats, es defineixen els casos de prova. Si imaginem una pila molt senzilla, el diagrama d'estats tindrà l'estructura següent.

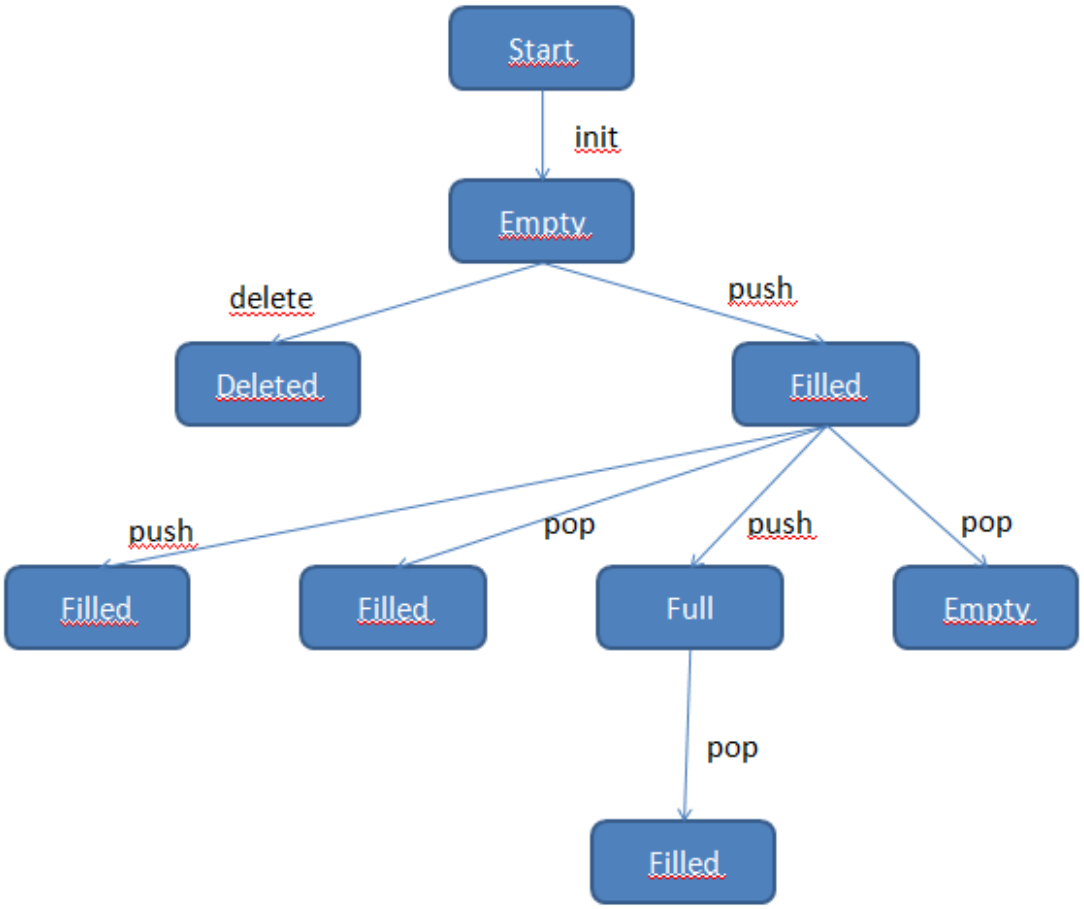


Fig. 2.5 Arbre de transició d'estats d'una pila

Cada branca de l'arbre és un cas de prova. A la Fig. 2.5 només apareixen casos nominals, es podria expandir l'arbre de manera que de cada estat en sortís una acció que portaria a un estat d'error. Per exemple, es podria comprovar que si la pila està plena i realitzem l'acció delete, el programa no borra la pila si no que ho controla o avisa a l'usuari. La prova es dóna per finalitzada quan cada estat i cada transició han estat executats en l'ordre correcte.

Depenent de l'estil de l'objecte sota test, aquest pot ser un bon mètode. Si l'objecte té estats que es defineixen amb moltes variables això pot induir a una gran càrrega de treball i a una complexitat que ens hauria de fer inclinar cap a un altre mètode de caixa negra. Aquest tipus de test només verifica que no existeixen errors en la seqüència d'estat, però no ens indica si hi ha errors dins els estats o de qualsevol altre tipus.

L'últim dels mètodes de caixa negra és el que està basat en casos d'ús. El client el que realment pacta normalment són els casos d'ús estil UML. Aquest mètode consisteix simplement en avaluar la interacció entre usuari i sistema, mentre aquest porta a terme un dels casos d'ús. Si el cas d'ús es compleix el test es dóna com a vàlid.

L'avantatge de les proves basades en casos d'ús és que són el tipus de prova ideal per a les proves de sistema i/o acceptació. L'inconvenient és que la informació que aporta no és suficientment completa per a definir totalment tot el cas de prova (jocs de dades, criteris de sortida, etc.).

Tots els mètodes de caixa negra tenen com a finalitat testar la funcionalitat del sistema. Al desconèixer com està programat el bloc, la documentació ha d'estar molt ben especificada. Una mala especificació portarà al disseny d'un test erroni i a un possible error que passi sense que hagi estat detectat. Tanmateix, tot allò que no estigui documentat el provador ho desconeix i per tant no ho pot testar. Els desavantatges de la tècnica de caixa negra no han aconseguit desbancar-la de ser la tècnica més utilitzada, ja que l'adequació funcional del sistema és l'objectiu de proves més important.

2.3.2.3. Altres mètodes

Hi ha més tècniques dinàmiques que les explicades anteriorment. Caixa blanca i caixa negra són només les més utilitzades. Existeixen tècniques creades a partir d'aquestes dues, i d'altres que no tenen res a veure.

El mètode de caixa gris, és una barreja entre les dues principals tècniques, caixa blanca i caixa negra. El que permet és que el dissenyador de tests té el coneixement de l'estructura interna de l'objecte de proves, però a l'hora de la execució les proves són funcionals, com si fos de caixa negra. És particularment útil per determinar valors límits, casos d'error, fent servir enginyeria inversa. En el cas que l'aplicació treballi amb arxius auxiliars, estil bases de dades, arxius de tipus log, etc. també es considera caixa gris si per realitzar els tests s'ha de modificar d'alguna manera aquests arxius auxiliars o la ruta establerta en el codi. Dóna més facilitat al provador ja que pot observar l'estat de l'objecte o objectes auxiliars, un cop ha finalitzat el joc de proves.

Les proves exploratòries segons Cem Kaner, qui va encunyar el terme per primer cop, són: un estil de test de software que emfatitza la responsabilitat i la llibertat personal del provador perquè contínuament optimitzi la qualitat de la seva feina tractant l'aprenentatge, l'execució i la interpretació de resultats de test com activitats mútuament de suport al llarg del projecte (veure [5]). Quan es realitzen tests exploratoris els resultats esperats no es coneixen fins que no s'executa el test per primer cop. Alguns poden ser previstos, altres difícilment. És el provador qui investiga si els resultats són correctes. Així doncs la qualitat de les proves recau en l'habilitat del provador per dissenyar els casos de prova i en la seva habilitat per a trobar defectes. L'avantatge principal d'aquesta tècnica és que es necessita molta menys preparació que en les altres. Un altre dels avantatges és segons l'afirmació, que només proves noves donaran resultats nous, que hi ha més possibilitats de trobar defectes diferents. Algun dels inconvenients és que els tests no es poden repassar per si tenen algun error, ja que es dissenyen i executen al mateix moment. Un altre desavantatge és que és més difícil traçar el cas de prova i el número de casos de prova pot variar molt d'un provador a un altre.

Les proves ad hoc són les menys formals de totes. És dissenyen i executen sense cap tipus de planificació ni documentació, totalment improvisades. El principal avantatge és que els errors importants es detectaran fàcilment. Ara bé, com que el test és improvisat i no està documentat és possible que sigui difícil de tornar a reproduir. Es pot pensar que les proves ad hoc són una versió descafeïnada de les proves exploratòries.

CAPÍTOL 3. EXEMPLE DE PROJECTE DE TEST: VERIFICACIÓ SCET-M

El centre espacial guaianés (CSG) és un dels ports espacials més grans que existeixen, es troba en territori de la Guaiana Francesa, i serveix als europeus com a base d'operacions principal. A més, disposa d'unes qualitats que la fan única, proximitat a l'equador i capacitat de llançadors diferents, entre d'altres.

La base la comparteixen la ESA i el CNES, que és el propietari original. Per a cada missió tots els treballadors de la base entren en frenesí dies abans del llançament, perquè tot surti el millor possible. Cada treballador dins de la seva secció col·laborarà en l'èxit del client i la missió.

Una d'aquestes seccions sota la subdirecció d'operacions del CNES és el Servei d'Aquisició de Mesures. El responsable d'aquest servei ha de gestionar els quatre dominis que el formen: la telemesura, la localització, la meteorologia, i l'anàlisi del sistema de mesures.

La telemesura està al servei dels subsistemes de localització, de salvaguarda de vol i d'Arianespace, que és la comercialitzadora dels llançaments Ariane. Arianespace demana el màxim de paràmetres possibles a la telemesura per assegurar-li als seus clients el correcte desenvolupament de la missió. A la salvaguarda de vol li interessa la telemesura per controlar constantment la disponibilitat de la telecomanda per neutralitzar el llançador, i també per corroborar la informació amb el sistema de localització. Existeix un sistema de localització apart de la telemesura. Encara que les dues tracten els mateixos paràmetres aquests s'obtenen de maneres diferents, i es comparen.

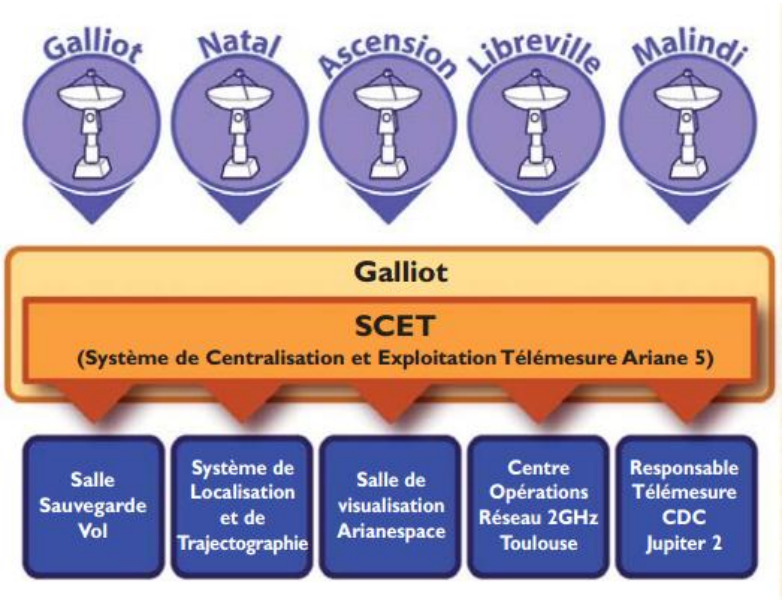


Fig. 3.1 Esquema de la xarxa amb les estacions de recepció

El sistema SCET (Système de Centralisation et Exploitation Télémesure) és el sistema de terra responsable de la telemesura pel que fa als Ariane 5. La part principal d'aquest sistema està instal·lat a l'estació Galliot, que es troba a uns quinze quilòmetres del centre tècnic. Galliot disposa d'antenes que són capaces de rebre la telemesura directament del llançador durant nou minuts aproximadament des de que s'enlaira. A partir d'aleshores, la telemesura és captada per altres estacions del mateix estil que estan situades al llarg de la traça de la seva trajectòria.

La plataforma SCET està formada per diferents llocs de treball, on en cada llançament hi ha una persona monitoritzant els paràmetres més importants, a la qual el software li permet treballar amb els paràmetres en temps real. No tots els llocs de la plataforma estan a Galliot, les dades s'envien a temps real a diferents indrets. Un dels llocs és la sala Jupiter on es troben els clients d'Arianespace durant el vol, aquí hi ha set dels vuit llocs de visualització. Així els clients poden veure els paràmetres de la telemesura en tot moment. L'altre sala on hi ha màquines SCET és a la sala de salvaguarda que n'hi ha dues. La sala de salvaguarda (SVG - Sauvegarde) és on treballen els únics treballadors que tenen responsabilitat civil. La seva funció és la de comparar les dades del sistema de localització amb les dades de la telemesura, i en cas d'algun error decidir quan neutralitzar el llançador. El darrer lloc on es pot trobar el sistema SCET és a la sala de pilotatge (PILO - Pilotage), aquí hi ha 5 màquines, una de visualització, els dos llocs de pilotatge i els dos llocs de preparació, veure Fig 3.2.

A més el software, prèviament al llançament, pot fer el que es diu la preparació de vol, que consisteix en introduir tots els paràmetres de la propera missió i comprovar que tot sigui correcte. Per a oferir una major confiança als directors de missió, també et permet efectuar una simulació del vol amb la preparació que es vulgui.

L'any 2008, el CNES va confiar amb Dassault Aviation, qui a la seva vegada es va associar amb GTD, per adaptar el sistema SCET a que també es puguin realitzar llançaments amb el llançador Vega, l'altre llançador europeu. Es pot fer servir el mateix software fàcilment, ja que l'estructura dels missatges de telemesura dels Vega és la mateixa que la dels Ariane 5. També es va aprofitar per renovar les tecnologies al darrere del sistema, concretament Ethernet per la comunicació entre llocs de treball, i afegir noves especificacions. S'espera així una major integritat dels missatges. Es va deixar oberta la possibilitat d'expandir el contracte i adaptar el sistema als llançadors russos Soyuz que també operen des del CSG.

Aquest nou sistema es va batejar SCET-M, Système de Centralisation et Exploitation Télémesure Multilanceur.

A continuació, s'explica l'estratègia de test de software que vam decidir seguir en aquest projecte i se'n mostren alguns exemples. Concretament, s'explica les proves unitàries i algunes proves estàtiques.

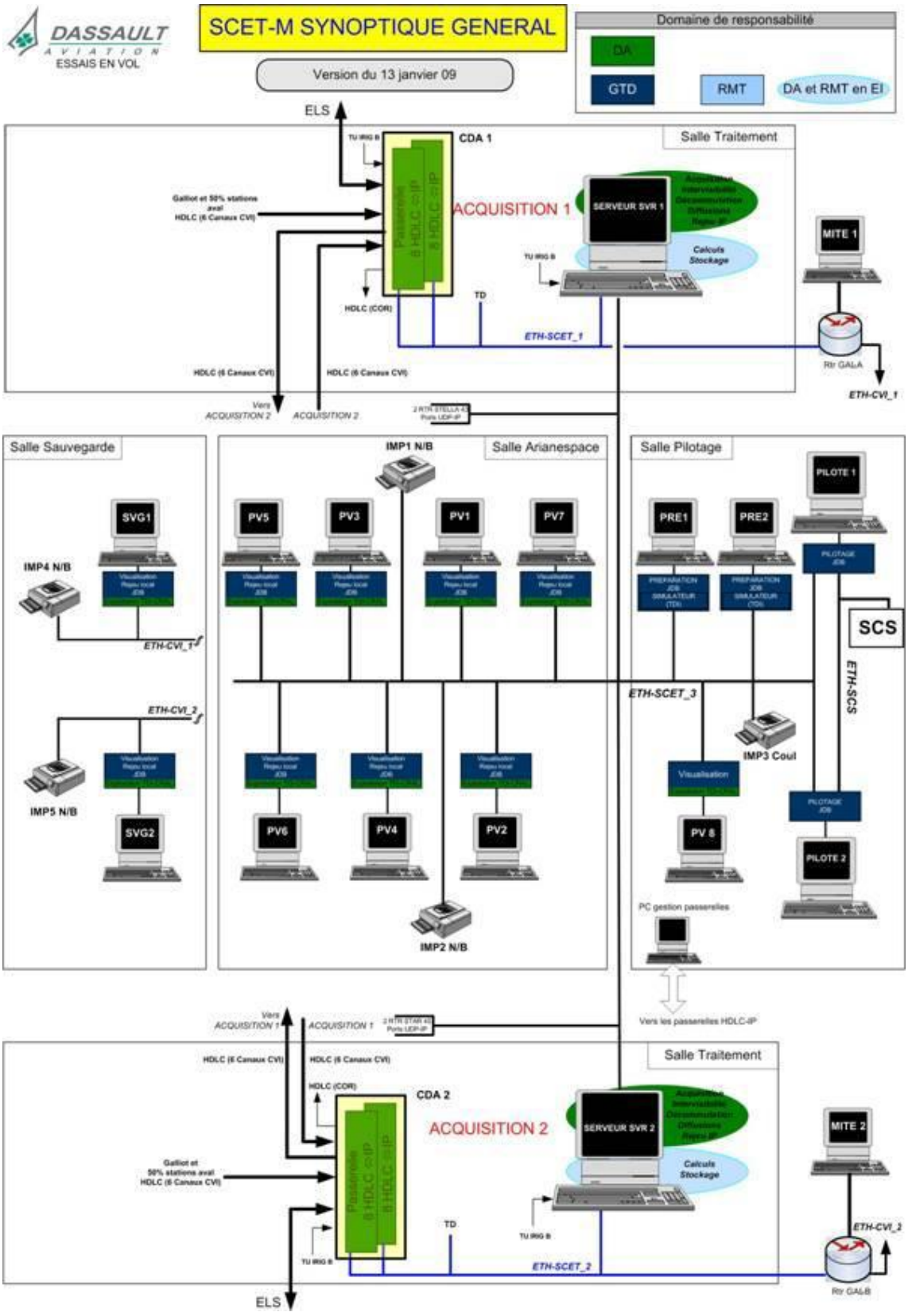


Fig. 3.2 Esquema general del sistema SCET-M

3.1. Estratègia del projecte de tests unitaris

SCET-M és un software programat en diferents llenguatges. El codi més antic és en C i en Ada, mentre que la gran majoria està en C++. També hi ha alguns mòduls de JAVA. La llibreria utilitzada és Qt versió 4.5.3 . Després existeix tota una sèrie de bases de dades i scripts de configuració, que són del projecte però no formen part del software pròpiament dit, entre d'altres raons perquè no són fitxers compilats. La plataforma corre sobre una distribució de Linux Red Hat 5.5.

L'equip de test disposa d'una altre màquina a la plataforma, per a poder executar les proves sense perjudicar els recursos de desenvolupament. Al tractar-se d'un projecte amb codi importat i un projecte entre empreses, es va decidir, diferenciar el que eren els mòduls crítics dels que no ho eren i separar el projecte de test en dos.

Per als mòduls crítics es va decidir fer servir caixa blanca. L'objectiu era aconseguir la màxima cobertura en el nivell de decisions per a cada classe crítica (component). Les classes crítiques són principalment les que regeixen la comunicació entre les màquines del sistema. Els canvis de protocol cap a Ethernet són una innovació respecte al software antic. Tot i així, hi ha altres classes que no tenen a veure amb la comunicació i també són crítiques.

Pel que fa referència a tot el que era codi nou, es va decidir testar funcionalment, per tant, mitjançant la tècnica de caixa negra. L'ús d'stubs es reserva només per als casos on l'automatització sigui impossible o el grau de dependències sigui molt alt, així s'assegura més funcionalitat. L'equip de proves però, té accés al codi font i a la documentació, així doncs, és realment un test de caixa gris.

Queda en mans del provador dissenyar els casos de test i els criteris de sortida, però sempre pot demanar la col·laboració al desenvolupador, de fet ningú coneixerà el codi millor que el desenvolupador. El provador mateix ha d'executar el test i observar els resultats. En cas que hi hagi resultats negatius, ha d'avisar al desenvolupador corresponent i entre els dos buscar una solució. És el desenvolupador qui finalment implementarà la solució en el repositori del projecte.

3.2. Implementació del projecte de tests unitaris

En els dos casos es fa servir l'eina d'IBM Rational Test RealTime (RTRT), versió 7.5. És una eina molt potent que a més de cobertura i tests funcionals, permet als provadors realitzar informes de memòria i anàlisi en temps real, apart incorpora altres característiques dels entorns de desenvolupament.

Una característica molt útil que inclou el programa de test és la d'afegir comandes de sistema a la línia d'execució. És útil perquè si el test ha de ser automàtic, interessa tornar a preparar l'estat original de l'objecte de test o de

les dades d'entrada per tal de poder repetir el test sense problemes. L'equip de test va desenvolupar uns scripts en llenguatge bash, que creaven els fitxers amb la configuració i amb el joc de proves desitjats.

També hi havia el problema de les dependències. Quan el projecte de test es descarregués en un altre directori, havia de funcionar igual. La solució va ser afegir una variable d'entorn als scripts d'arrencada on s'hi inicialitza el directori on es troba el codi. El codi en el projecte de test i altres arxius precompilats estan sempre en funció d'aquesta variable.

La manera de fer servir l'eina o crear el projecte de test de caixa negra va ser anàloga a l'organització del projecte. Igual que el sistema es divideix en subsistemes i a la seva vegada en mòduls, el projecte de test es divideix en subprojectes de test. Cada mòdul de subsistema del projecte representa un subprojecte de test, i cada classe d'aquest mòdul té el seu joc de proves. A cada subprojecte de test l'hi correspon una carpeta, que és on l'eina de test treballa i genera els arxius precompilats necessaris per a executar el test.

Dins d'aquesta carpeta però també hi ha d'haver uns altres tres tipus d'arxius. Aquesta és la raó de l'existència de les altres tres carpetes dins de la del subprojecte de test. La que fa referència a les dades d'entrada i sortida (dataSet), la dels arxius que fan referència als resultats de les proves(reports), i una per als arxius que l'eina necessita però que el provador ha de programar o dissenyar abans(Resources)(veure Fig. 3.3).

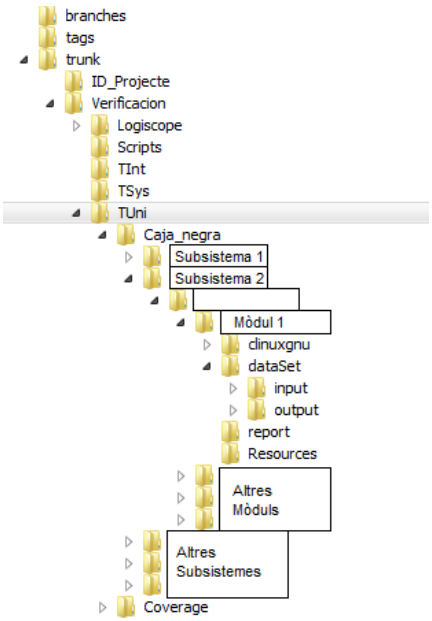


Fig. 3.3 Arborescència del projecte de test

A l'hora de decidir l'arborescència del projecte de proves es va optar per reproduir la mateixa del projecte fins a cada mòdul de subsistema i aleshores

apareixen les pròpies carpetes de test.. Així el nom del subprojecte de test és el mateix que el del mòdul. Aquesta és una bona manera d'organitzar, perquè degut al gran volum del projecte i al número d'arxius addicionals és molt fàcil equivocar-se i mesclar recursos o dades de proves diferents.

Pel projecte de test de caixa blanca el sistema d'organització és molt semblant al de caixa negra. La diferència principal és que no reprodueix l'arborescència del projecte, sinó que simplement s'engloben en quatre mòduls depenent del subsistema on es troben. Una altre diferència és que no existeix la carpeta del dataSet, com que només s'analitza cobertura no és necessària.

L'RTRT et permet tenir moltes configuracions diferents de compilador, linker, etc., per a cada joc de casos de test. Els tests no canvien encara que l'entorn sí, això descobreix regressions ràpidament. I permet exportar el projecte i executar-lo en altres màquines amb configuracions diferents sense masses problemes.

L'eina de test instrumentalitza el codi de manera que hi afegeix codi propi al codi font analitzat per a fer les seves comprovacions. Un cop es compila i s'executa el test les dades es bolquen en arxius en format xml, html o sobre la interfície dels informes de resultats.

Per tenir-ne una idea, bàsicament el que fa és crear una classe test que és filla de la classe sota anàlisi. I afegeix les seves funcions de control. Depenent de la prova, com que necessita el codi font i arxius precompilats en release, a més dels quatre arxius de recursos, l'execució pot arribar a ser molt lenta. Per evitar això es poden afegir stubs.

Els quatre arxius que necessita cada test de component són els de la carpeta Resources. El primer és el .otc i defineix els contractes de les funcions, és a dir, et permet testar les taules d'estats i invariants de cada mètode de la classe. L'arxiu .otd és el més important. És on es programen les proves, aquí apareixen el jocs de proves amb el seus casos de test, tant nominals com d'error. El fitxer .stb és el que conté els stubs. I finalment el .dcl és el de declaracions, aquest té el propòsit de redefinir variables i definir dependències. El propi RTRT té una eina que permet generar els quatre arxius en funció de la classe de test, però seguint un perfil base.

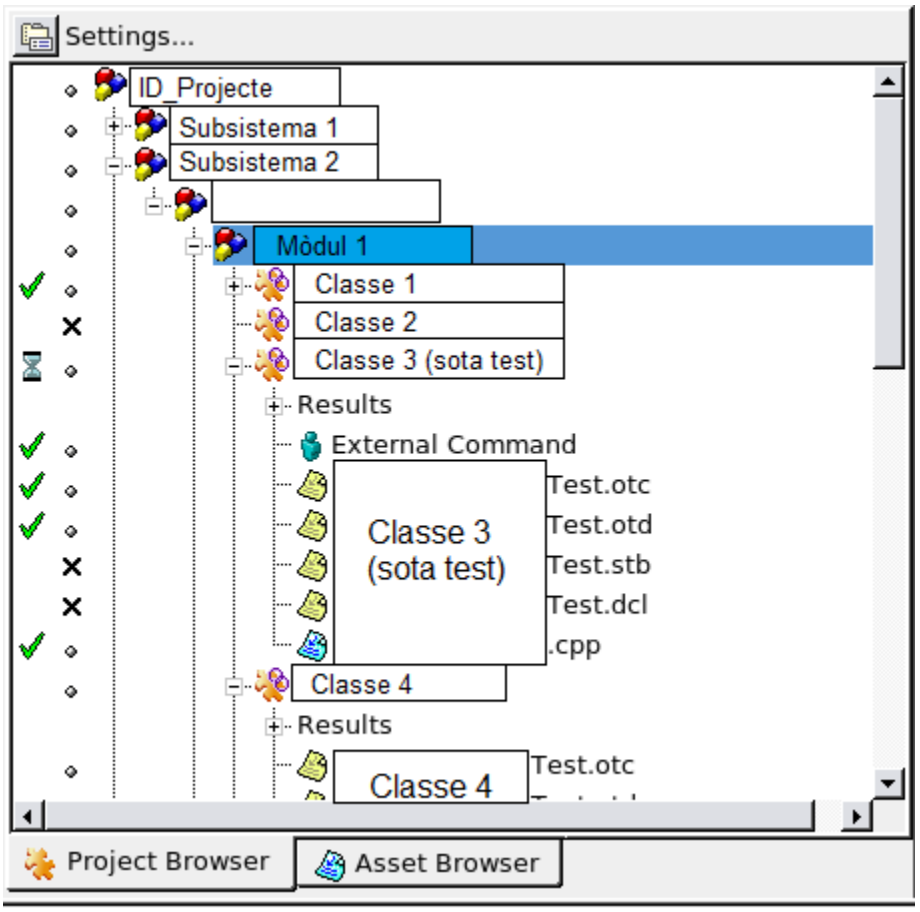


Fig. 3.4 Explorador de l'RTRT amb el projecte de test

L'arxiu .otd, test driver, és el més important perquè és realment on s'implementa el disseny del test. Aquí han d'aparèixer tots els casos de test planificats. El llenguatge és una variació del c, però sempre hi ha l'opció inserir-hi codi font. La manera d'estructurar les proves en el fitxer és per jerarquia, hi ha test class, test suite i test case. Cada un engloba el següent.

Tal i com es mostra a la Fig. 3.5, la característica principal dels tests class és que estan formats per dues parts. La primera és el marc on aniran els nivells de test menor i la segona està situada al final de l'arxiu i conté els noms dels diferents jocs de proves, test suite en aquest cas, segons l'ordre que s'executaran.

El nivell de test suite està format per 3 parts diferenciades: el pròleg, l'epíleg i els diferents casos de test. Al pròleg és on es defineix l'objecte de test que el programa haurà de monitoritzar, també és útil per definir variables auxiliars o preparar les dades d'entrada. L'epíleg té la funció de destruir l'objecte i anàlogament amb el pròleg, és on es pot fer algun preparatiu abans de destruir-lo. Cada test suite representa un objecte de test. Tot el que es programi en un test suite té visibilitat només dins del test suite.

```
// Test class
TEST CLASS Test
{
	TEST CLASS TestAlphabet (Alphabet)
	{
		PROLOGUE
		{
			// Declarations of variables needeed by this test class.
			// Actions to be performed before executing this test
			// class.
		}

		TEST SUITE testA
		{
		PROLOGUE
		{
			# Alphabet obj0;
		}
		TEST CASE nominal
		{
			PRINT ("enter in testA");
			#{
				bool result = obj0.A();
			}#
			CHECK (result);
			PRINT ("exit from testA");
		}
		EPILOGUE
		{
		}
		}
		.
		.
		.
		EPILOGUE
		{
			// Actions to be performed when leaving this test class.
		}

		RUN
		{
			// By default, all test case are called here.
			testA;
			testB;
			testC;
			testD;
			testE;
		}
	}
	RUN
	{
		TestAlphabet (Alphabet);
	}
}
RUN
{
	Test;
}
```

Fig. 3.5 Exemple de fitxer .otd

Els test case s'executen per ordre d'aparició. És aquí on realment hi ha la feina de programar la prova i pensar la millor manera d'avaluar. Les verificacions es duen a terme mitjançant la paraula clau CHECK, l'expressió a avaluar ha de ser equivalent a un booleà. També hi ha CHECK METHOD que el què fa és comprovar si en algun moment d'aquell test case s'ha cridat com a mínim un cop una funció en concret. Aquestes són les úniques maneres d'obtenir uns informes sobre la funcionalitat de la prova.

3.3. Cas de test unitari de caixa negra

Aquest subapartat té la intenció d'entrar en detall explicant els tests unitaris que apareixen a la Fig. 3.6. Tal i com s'observa, els test suite engloben diversos test cases. Normalment per a cada funció n'hi ha un pel cas nominal i un altre pel cas d'error. Aquest test forma part d'un test class que comprova les funcionalitats d'una classe que tracta amb fitxers xml. Forma part del subsistema de simulació i treballa amb els paràmetres de funcions que es necessiten durant la simulació.

La funció a provar el que fa concretament és afegir un node de tipus simParams dins de l'objecte de Qt que simula virtualment l'organització del xml (DOM). Però no reescriu el fitxer xml original. Aquest node és d'alt nivell dins l'estructura xml (segon nivell). El node germà és un de tipus simTrames, però és una altra funció qui s'encarrega.

En el procés de disseny del test, es va decidir que el criteri de sortida seria si la funció ha inclòs correctament el node simParams dins l'objecte DOM. D'aquesta manera, el criteri de sortida pel cas d'error és que no s'hagi afegit cap node d'aquest tipus.

A l'hora d'implementar la prova es van escollir com a paràmetres d'entrada dos arxius suposadament xml. El primer està destinat al cas nominal i conté la capçalera típica dels xml i el node pare, de manera que la funció hauria d'interpretar el fitxer sense problemes i carregar el node dins de l'objecte virtual. En el segon cas d'error es va servir un arxiu amb extensió d'xml però realment el fitxer està buit. Així la funció hauria de fallar ja que no pot interpretar l'estructura i crear l'objecte DOM.

Per a aconseguir llançar el test s'ha de preparar els objectes de la classe perquè tinguin un estat que permeti cridar la funció. En aquest cas es tracta de carregar un fitxer a l'objecte DOM de la classe. Aquests fitxers es creen especialment pel test. El del cas nominal es diu test_addSim.xml i el del cas d'error test_addSim_error.xml. Tant un com l'altre es troben a la carpeta del joc de dades d'entrada ("./dataSet/input/").

Un cop relacionades les dades d'entrada del test amb l'objecte testat, es procedeix a avaluar que l'estructura xml no contingui ja un node de tipus simParams. Del test es podria concloure erròniament que la funció ha afegit el node al DOM, si el fitxer ja contenia algun node d'aquest tipus abans d'executar

el test. Per tant, un cop està la classe en l'estat desitjat es comprova que efectivament no hi ha cap node com aquest dins del DOM.

```
TEST SUITE testaddSimParamXMLnode
{
PROLOGUE
{
      # FichierDefinition obj0;
}
TEST CASE nominal
{
      PRINT ("enter in testaddSimParamXMLnode_nominal");
      #{
            QString string("./dataSet/input/test_addSim.xml");
            QFile file(string);
            obj0.xmlFile.setFileName(string);
            obj0.xmlDom->setContent(&file);
      }#
      CHECK (!obj0.xmlDom->toString().contains("simParams"));
      #{
            obj0.addSimParamXMLnode();
      }#
      CHECK (obj0.xmlDom->toString().contains("simParams"));
      PRINT ("exit from testaddSimParamXMLnode_nominal");
}
TEST CASE error
{
      PRINT ("enter in testaddSimParamXMLnode_error");
      #{
            QString string("./dataSet/input/test_addSim_error.xml");
            QFile file(string);
            obj0.xmlFile.setFileName(string);
            obj0.xmlDom->setContent(&file);
      }#
      CHECK (!obj0.xmlDom->toString().contains("simParams"));
      #{
            obj0.addSimParamXMLnode();
      }#
      CHECK (!obj0.xmlDom->toString().contains("simParams"));
      PRINT ("exit from testaddSimParamXMLnode_error");
}
EPILOGUE
{
}
}
```

Fig. 3.6 Exemple de test nominal i d'error

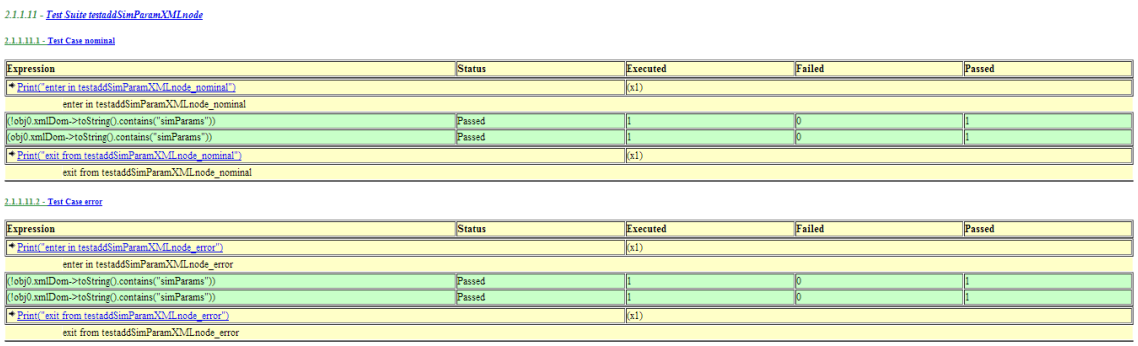
Seguidament s'executa la crida de la funció. Un cop la funció ha acabat, el programa de test procedeix a avaluar l'estat dels objectes. Per això en el test apareix una altra sentència a verificar amb la paraula clau CHECK just abans d'imprimir que el test ha finalitzat en els resultats.

A l'epíleg no hi ha res programat perquè només s'ha de destruir l'objecte que s'ha declarat prèviament al pròleg.

Finalment, un cop acabada l'execució del joc de proves, l'RTRT genera uns arxius de tipus xrd que permeten visualitzar els resultats. La Fig. 3.7 mostra els resultats d'una manera clara. En verd apareixen totes les verificacions (CHECK) que s'han programat al fitxer otd i s'han avaluat positivament. En vermell apareixerien les que no s'han complert. I en groc en el cas que no s'hagi arribat a executar la verificació. En aquest cas, el test ha pogut verificar correctament el comportament d'una funció tant en un cas nominal com en un cas d'error.

En els informes de resultats també es permet introduir-hi comentaris o valors dinàmics per acabar d'ajudar a comprendre què s'ha dut a terme.

Un cop finalitzada l'execució del test class si els informes apareixen tots en verd voldrà dir que tot funciona segons el que s'esperava i es pot passar a testar una altra classe.



2.1.1.11 - Test Suite testaddSimParamXMLnode

2.1.1.11.1 - Test Case nominal

Expression	Status	Executed	Failed	Passed
Print("enter in testaddSimParamXMLnode_nominal")		(x1)		
enter in testaddSimParamXMLnode_nominal				
(!obj0.xmlDom->toString().contains("simParams"))	Passed	1	0	1
(obj0.xmlDom->toString().contains("simParams"))	Passed	1	0	1
Print("exit from testaddSimParamXMLnode_nominal")		(x1)		
exit from testaddSimParamXMLnode_nominal				

2.1.1.11.2 - Test Case error

Expression	Status	Executed	Failed	Passed
Print("enter in testaddSimParamXMLnode_error")		(x1)		
enter in testaddSimParamXMLnode_error				
(!obj0.xmlDom->toString().contains("simParams"))	Passed	1	0	1
(!obj0.xmlDom->toString().contains("simParams"))	Passed	1	0	1
Print("exit from testaddSimParamXMLnode_error")		(x1)		
exit from testaddSimParamXMLnode_error				

Fig. 3.7 Informe de resultats del test de la Fig. 3.6 Exemple de test nominal i d'error

3.4. Cas de test unitari de caixa blanca

Aquest subapartat té la intenció d'entrar en detall explicant el test unitari que apareix a la Fig. 3.8. La funció de l'exemple forma part d'una classe que gestiona el necessari per crear una nova sessió en un dels llocs de visualització. La funció en concret el que realitza és la recepció del missatge de canvi de mode i mira si aquest canvi és possible. En cas positiu, actualitza els estats de comunicació i el realitza. Finalment, el mètode crea un altre missatge que informa a tot el sistema del canvi d'estat.

A diferència dels de caixa negra com que no es prova la funcionalitat no hi ha la necessitat d'agrupar els mètodes en test suites. L'objecte de test es defineix només a nivell de test class i cada test case fa servir el mateix. Per cada

branca de la funció ha d'existir un test case així s'assolirà el cent per cent de cobertura.

L'ús de stubs està molt generalitzat. L'exemple mostra dos casos de la mateixa funció i un dels stubs que s'utilitzen. Es pot apreciar que els paràmetres de la funció no canvien d'un test a l'altre. La manera com s'aconsegueix que l'execució recorri les diferents branques és mitjançant els stubs.

```
TEST CASE testtraitementMsgMode_3
{
      CHECK STUB MsgCmdNouveauMode_getModeStub{
            #mode = Types::MODE_HC;
      }
      PRINT ("enter in testtraitementMsgMode_3");
      // You have to adapt the code below for your test.
      #{
            quint16 id;
            MsgCmdNouveauMode msg;
            obj0.traitementMsgMode(id, msg);
      }#
      PRINT ("exit from testtraitementMsgMode_3");
}

TEST CASE testtraitementMsgMode_5
{
      CHECK STUB MsgCmdNouveauMode_getModeStub{
            #mode = Types::MODE_ESS;
      }
      CHECK STUB QString_endsWithStub{
            #b=true;
      }
      PRINT ("enter in testtraitementMsgMode_5");
      // You have to adapt the code below for your test.
      #{
            quint16 id;
            MsgCmdNouveauMode msg;
            obj0.traitementMsgMode(id, msg);
      }#
      PRINT ("exit from testtraitementMsgMode_5");
}
//EXEMPLE D'STUB NO PERTANY A .otc
//STUB MsgCmdNouveauMode_getModeStub : const quint32
//MsgCmdNouveauMode::getMode()const
//{
//    #quint32 mode;
//    ...
//    #return mode;
//}
```

Fig. 3.8 Exemple de test de cobertura i stub

Durant l'execució si hi ha la crida a una funció stubada podem modificar el comportament de l'stub fent servir les paraules clau CHECK STUB. Així per a cada test l'stub simularà un comportament diferent.

No s'ha de programar res addicional per a realitzar un anàlisi de cobertura, el mateix programa de test ja el realitza en cada execució. En aquest cas només es verifica la cobertura a nivell de decisions.

A la Fig. 3.9 es pot comprovar els resultats del test. Simplement és la quantitat de branques que ha recorregut sobre el total.

RTRT fa servir el mateix codi de colors per caixa blanca i caixa negra. Així un cent per cent de cobertura equival a tota la columna corresponent de color verd. També té la opció de representar-la gràficament mitjançant un diagrama de barres. Per tant, la classe testada a l'exemple compleix el requisit de cobertura sense excepcions.

1 - Summary

Root	
Functions	100.0% (40/40)
Functions and exits	100.0% (80/80)
Statement blocks	100.0% (103/103)
Decisions	100.0% (119/119)

2 - Files list

Display absolute values only.

Item ▼	Functions	Functions and exits	Statement blocks	Decisions
TrtCmdNouvelleSession.cpp	17/17	34/34	40/40	48/48
void FunctionA ()	1/1	2/2	1/1	1/1
void FunctionB ()	1/1	2/2	1/1	1/1
void FunctionC ()	1/1	2/2	1/1	1/1
void TrtCmdNouvelleSession::traitementMsgMode (quint16, MsgCmdNouveauMode &)	1/1	2/2	11/11	13/13
void FunctionD(D)	1/1	2/2	10/10	14/14
TrtCmdNouvelleSession & TrtCmdNouvelleSession::TrtCmdNouvelleSession ()	1/1	2/2	1/1	1/1
TrtCmdNouvelleSession & TrtCmdNouvelleSession::TrtCmdNouvelleSession (S*)	1/1	2/2	1/1	1/1
TrtCmdNouvelleSession & TrtCmdNouvelleSession::TrtCmdNouvelleSession (TrtCmdNouvelleSession &)	1/1	2/2	1/1	1/1
void TrtCmdNouvelleSession::~TrtCmdNouvelleSession ()	1/1	2/2	1/1	1/1
void FunctionE (E)	1/1	2/2	1/1	1/1
S * FunctionF ()	1/1	2/2	1/1	1/1
const quint32 FunctionG () const	1/1	2/2	1/1	1/1
const QString FunctionH () const	1/1	2/2	1/1	1/1
const QString FunctionI () const	1/1	2/2	1/1	1/1
bool FunctionJ(J)	1/1	2/2	1/1	1/1
TrtCmdNouvelleSession & FunctionK (K)	1/1	2/2	2/2	3/3
void FunctionL (L)	1/1	2/2	4/4	5/5

Fig. 3.9 Informe de resultats dels tests de la Fig. 3.8 Exemple de test de cobertura i stub

3.5. **Test Estàtic**

Al tractar-se d'un projecte software de gran criticitat, el CNES s'assegura que tot el codi tingui una uniformitat i uns estàndards addicionals al que marquen les normes de qualitat internacional. Així, imposa per requisit el cumpliment d'unes normes a través d'una eina de proves estàtiques.

IBM Rational Logiscope, versió 6.6, és el software utilitzat. L'aplicació té el seu entorn gràfic a més de la possibilitat fer-lo servir a través d'un terminal amb línia de comandes, aquesta última opció obre moltes possibilitats. Aquesta eina permet fer un anàlisi de totes les mètriques del codi i també té unes regles implementades, que es poden reprogramar o afegir-ne de noves sempre que es vulgui, per verificar el codi. Logiscope té moltes més opcions com ara anàlisi de cobertura, generació de diagrames, però pel que fa referència a SCET-M només es fa servir per les dues esmentades anteriorment, anàlisi de mètriques i de regles.

Taula 3.5. Exemples de mètriques comunes i de C++

Nivell	C++/Comú	Mètrica
Tots	Comú	Sum of cyclomatic numbers
Tots	Comú	Number of lines of comments
Tots	Comú	Number of statements
Tots	Comú	Number of similarities
Funció	Comú	Total number of operands
Funció	Comú	Number of directcalls
Funció	Comú	Cyclomaticnumber
Funció	Comú	Maximumnestinglevel
Tots	C++	Attributeinheritance factor
Aplicació	C++	Inheritancetreecomplexity
Classe	C++	Number of local constants
Funció	C++	Number of gotos
Mòdul	C++	Number of preprocessorstatements

Les mètriques són característiques parametritzades del codi, n'hi ha de nivells i tipus diferents. A la Taula 3.5.Exemples **de mètriques comunes i de C++**, apareixen algunes de les mètriques amb les que es treballa. El que fa Logiscope és comparar aquestes mètriques amb els valors que demana el client, i generar uns informes en format html. Els informes indiquen un criteri de qualitat entre els quatre que hi ha (perfect, good, fair i poor) en funció del tant per cent d'arxius que han complert els requisits. Òbviament també et diu en el cas que algun no s'hagi complert, en quina classe o funció, depenent del nivell, es troba.

El que s'aconsegueix amb aquest procediment és detectar sobretot el paràmetre de qualitat de la mantenibilitat. Com més comentaris té un codi per exemple, més fàcil és entendre'l i modificar-lo en cas que s'hagi de fer.

Les regles de programació a diferència de les mètriques són enunciats que fan referència al disseny o a la qualitat del codi que el client vol. Algunes regles són semblants a les mètriques però no iguals. Algunes regles poden verificar el nivell d'alguna mètrica però mai al revés.

El que es pretén amb moltes de les regles és aconseguir un codi d'una gran formalitat, és a dir, amb una qualitat de mantenibilitat molt elevada. Ara bé, moltes altres tenen la missió de garantir que no hi hagi forats de seguretat o bé assegurar bones pràctiques de programar en benefici de la fiabilitat per exemple.

Al desenvolupador li pot semblar una càrrega excessiva haver de comprovar tantes regles, a més si en algunes d'elles no els hi troba el sentit. La veritat és però que es tracta d'un sistema crític i és el client qui verdaderament imposa el compliment d'aquestes normes.

La metodologia de treball respecte les proves estàtiques és lleugerament diferent a la metodologia emprada per les proves funcionals. No apliquen les mateixes regles per tots els fitxers. Cada desenvolupador s'encarrega de passar Logiscope i corregir el codi, com una part més del procés. Cal notar, que si es justifica davant del client, és possible ometre alguna regla en concret. Així resta en el criteri del desenvolupador decidir si la norma aplica o no.

L'aportació de l'equip de test a les proves estàtiques va ser crear els arxius que contenien la llista de regles. Per tal que tot l'equip les tingués preparades per quan el codi ja estigués quasi desenvolupat.

A més a més, l'equip de test va desenvolupar un script en bash que automatitza el procés d'anàlisi estàtic. Però no només l'automatitza. Degut a la diferenciació entre codi crític i no crític prepara diferents projectes de test pels dos tipus possibles. També els classifica en funció de si es codi nou o codi vell, i també en funció del llenguatge de programació. El codi nou són els fitxers completament nous o els fitxers que han sofert alguna modificació. Així l'script sap en cada moment quines mètriques i quines regles pertoquen a aquella peça de codi. I genera llistes d'arxius i l'arborescència adequada per organitzar correctament els informes.

Taula 3.6 Exemples de regles de programació

Nom:	Don.Enumerations
Explicació/Justificació:	No se permiten enteros literales en el código, exceptuando 0 y 1 y la declaración de constantes, enumerados y defines.
Nom:	Err.DestructeurToutes
Explicació/Justificació:	No se permiten destructores sin bloque try... catch a no ser que sean vacios.
Nom:	Don.Initialisation
Explicació/Justificació:	All variables must be initialized before they are used. Not all compilers give the same default values. Unexpected behaviour can be avoided with better control over variable values.
Nom:	Don.TypeBooléen
Explicació/Justificació:	Use real boolean expressions. The tests in control structures must contain proper boolean expressions.
Nom:	Pr.Aeration
Explicació/Justificació:	Se debe de dejar un espacio entre operador y operandos para los operadores binarios. Los operadores unarios estaran junto al operador.
Nom:	Tr.Booleen
Explicació/Justificació:	No se permiten expresiones boleanas complejas en las condiciones de las instrucciones if, while, for y do_while.
Nom:	Tr.BoucleSortie
Explicació/Justificació:	En un bucle no se permite la utilización de break o return.
Nom:	Tr.Destructeur
Explicació/Justificació:	Destructors of base classes must be declared virtual so destructors are called before memory deallocation
Nom:	Tr.ParReference
Explicació/Justificació:	No se permiten los parámetros tipo puntero. Se deben de pasar referencias.
Nom:	Tr.Parentheses
Explicació/Justificació:	Removes ambiguity about the evaluation priorities.

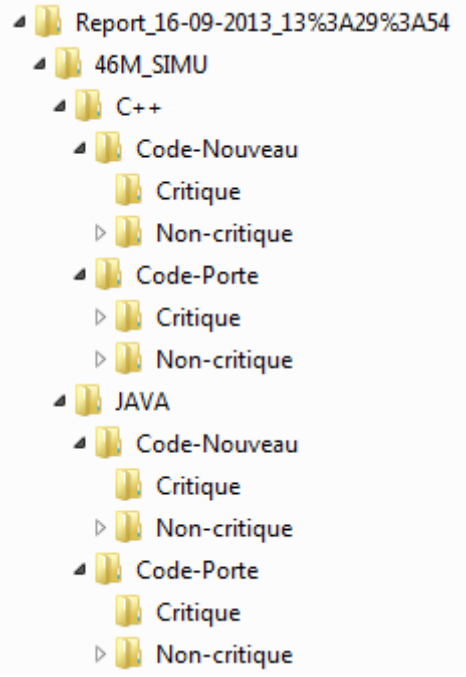


Fig. 3.10 Arborescència d'un informe d'anàlisi estàtic

Aquest script s'executa cada nit en la màquina de qualitat de la plataforma SCET-M. Cada nit es descarrega l'última versió del codi del repositori. Fa la separació en projectes en funció del llenguatges de programació, de la antiguitat i de la criticitat, veure Fig. 3.10. I abans de finalitzar l'execució, guarda els resultats de les proves on es vulgui i borra tots els arxius complementaris generats junt amb el codi.

Fig. 3.11 Plataforma SCET-M a GTD Barcelona

CAPÍTOL 4. CONCLUSIONS

Tot projecte de test té com a finalitat millorar la qualitat d'un producte software, reduir el risc de detectar errors o satisfer requisits imposats pel client.

Hi ha moltes estratègies diferents per testar software. La dificultat rau en escollir la combinació més adequada. Pot ser una estratègia més preventiva, dissenyant els tests abans. Pot ser una estratègia més reactiva, dissenyant els tests després. O bé analítica, dissenyant els tests basats en els riscos. El cap de projecte ha de decidir què escollir segons el seus objectius i els ha de deixar clars a la resta de l'equip. També ha de decidir quan donar per finalitzada la campanya de test.

La feina del provador és executar els casos de prova i registrar-ne els resultats. Moltes vegades també haurà de dissenyar i implementar els casos de test. Si hi ha cap error el provador ho ha de comunicar. El test es guarda per executar-lo més endavant per a verificar que el defecte o l'error han estat corregits.

A l'hora de dissenyar les proves és necessària una bona comunicació amb la resta de l'equip, especialment amb els desenvolupadors. Existeix la percepció que els provadors tenen una activitat destructiva. Quan un provador troba algun defecte és que ha tingut èxit. Molts cops és necessària una altra mirada per acabar d'obtenir el millor software.

Un bon provador en general ha de ser escèptic i atent a l'hora d'analitzar l'objecte de proves. A més, cal certa habilitat al traslladar males notícies als desenvolupadors. L'experiència també juga un paper clau al ajudar a identificar on poden aparèixer els errors. Aquestes característiques també les pot reunir un desenvolupador però aquest mai analitzarà la seva "creació" de manera imparcial.

El treball conjunt entre provadors i desenvolupadors és el que aconseguirà una activitat constructiva amb un producte software de qualitat.

La meva aportació en aquesta memòria del treball ha consistit en plasmar la visió d'un provador o enginyer de test sobre el procés de proves en general. Els exemples de casos pràctics formen part del projecte de test on la majoria de casos de test han estat dissenyats, implementats i executats per mi. S'afegeixen al document per tal de mostrar casos pràctics. L'automatització de tot el projecte de test ha suposat el major repte a l'hora d'implementar els tests.

De les explicacions de les característiques dels diferents projectes de test de SCET-M, se'n pot concloure que és realment impossible arribar a tenir-ho tot cent per cent testat. És massa aviat per valorar l'estratègia de test seguida, ja que encara s'està desenvolupant. Però de moment amb la fase 1 del projecte, que ja està entregada i operativa, i amb la seva corresponent campanya de test es pot quasi afirmar que s'ha aconseguit un codi lliure de defectes.

4.1. Principis del procés de proves software

Un test correctament executat no és indicador de qualitat, a menys que hagi sigut negatiu a l'anterior execució. Així la relació amb la qualitat software no és tant clara com semblava al principi. Només vol dir que el cas de test ha sigut incapaç de trobar un error.

Alguns mètodes de desenvolupament com el TDD o l'"agile" posen les proves al centre del desenvolupament. No es pot caure en l'error de pensar que un cas d'error pot substituir tots els casos, que és el que defineix l'especificació.

Degut a la pròpia naturalesa dels projectes software un error prèviament solucionat pot tornar a aparèixer. El cas de test que verifica que ja no existeix l'error, haurà de seguir formant part del projecte de test per evitar regressions.

A la pràctica no és realista intentar executar tots els possibles casos de prova, ni anar mirant manualment un per un quin ha estat el resultat. Així el procés d'execució i generació de informes ha d'estar automatitzat.

Només un provador és capaç de pensar en casos interessants de prova. Tota la resta de tests més obvis es pot deixar el seu disseny a càrrec de generadors de casos de test.

4.2. Cultura de la seguretat

La seguretat entesa com la confiança en la fiabilitat d'un sistema, ha de ser una actitud comú a tota la indústria aeroespacial.

Aquest sector no es pot permetre ser autocomplaent i pensar que perquè una cosa ha funcionat fins ara ho continuarà fent. Encara que els departaments de gestió estiguin preocupats pels riscs en la seguretat d'un sistema, normalment els equips de test i de qualitat són els primers en patir les conseqüències de les restriccions pressupostàries. Ja que se'ls hi assigna una baixa prioritat i es consideren les parts menys crítiques del projecte. Aleshores no es pot esperar la mateixa qualitat de producte.

Entendre els riscos associats al software és molt important. Molts accidents han tingut entre les seves causes una actitud de la direcció de missió infravalorant el software.

El software ens dóna el potencial d'incrementar moltíssim les nostres capacitats de control. Per contra, ens introdueix una sèrie de possibles causes d'error que requereixen ser preses en compte. S'ha d'aplicar les mateixes pràctiques que s'apliquen en altres branques d'enginyeria considerades més sèries, entenent les diferències i ajustant-les al procés de desenvolupament.

Només amb el millor treball conjunt de tots els sectors serà possible fer front als grans reptes tecnològics necessaris perquè l'ésser humà pugui seguir amb la seva aventura espacial.

4.3. **Estudi d'ambientalització**

Per a la realització d'aquest TFC l'impacte ambiental generat ha estat mínim ja que només s'ha involucrat software i juntament amb la redacció d'aquest document. La tecnologia emprada està bastant generalitzada. El cost d'aquest treball ha estat mínim també perquè s'han aprofitat recursos industrials. Per tant, es pot considerar un treball de final de carrera viable tècnica, econòmica i ambientalment.

Referències

[1] "Mars Climate Orbiter Mishap Investigation Board Phase I Report" November 10, 1999

[1] Report by the Inquiry Board, "Ariane 5 Flight 501 Failure" Nº 33-1996

[2] ISO/IEC 9126 and ISO/IEC 25010:2011

[3] http://junit.org/

[4] Colin Higgins, "Integration And System Testing", *School of Computer Science*, University of Nottingham

[5] Cem Kaner, "A Tutorial in Exploratory Testing", p. 36

[6] M. R. Woodward, M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC*", Information and Software Technology 48* (2006)

N.G. Leveson, "The Role of Software in Spacecraft Accidents", *MIT ESD Technical Paper*.

Keith J. Britton and Dawn M. Schaible, "Testing in NASA Human-Rated Spacecraft Programs: How much is just enough?", *System Design and Management Program*.

Earll M. Murman, Myles Walton and Eric Rebentisch, "CHALLENGES IN THE BETTER, FASTER, CHEAPER ERA OF AERONAUTICAL DESIGN, ENGINEERING AND MANUFACTURING", Massachusetts Institute of Technology, *Paper to Appear in The Aeronautical Journal*.

Nancy G. Leveson, "A Systems-Theoretic Approach to Safety in Software-Intensive Systems" *Aeronautics and Astronautics Department and Engineering Systems Division*, Massachusetts Institute of Technology.

"Probador Certificado Nivel Básico", Programa de estudios 2005 del ISTQB.

Santiago Ramírez de la Piscina Millán, "Medida de magnitudes", Técnicas experimentales.

Stéphane Rousseau, Gavin Walmsley, Jean-Claude Agnese, Jean-Claude Rubio and Jean-Luc Voyer, "Ariane 5 launch, first step of ATV's long trip to the ISS", AIAA 2010-2213.

"User guide" IBM Rational Test RealTime Version 7.5.

Centre Spatial Guyanais, "PRESENTATION GENERALE DU CNES/CSG", CG/SDO/OP/05/413.

ETH Zürich and Eiffel Software "Seven Principles of Software Testing" Bertrand Meyer,.

GTD, "GTD References in Space Ground Segment", Ago 2010.

Leslie A. (Schad) Johnson, DO-178B, "Software Considerations in Airborne Systems and Equipment Certification"

DO-178B and DO-178C, Software Considerations in Airborne Systems and Equipment Certification

Glossari

Agile: tipus de grup de models de desenvolupament, basats en un desenvolupament incremental i iteratiu, on els requisits i les solucions van evolucionant al llarg de la vida del projecte.

Arxius log: tipus de fitxers que registren tots els esdeveniments que ocorren durant l'execució d'un programa.

Coverage: es refereix a quant de codi s'executa durant un test en concret respecte el total. Equivalent a cobertura de codi, es pot donar en percentatge.

DOM: de l'anglès Document Object Model, es una interfície de programació d'aplicacions que permet representar documents HTML i XML.

Driver - controlador: regula l'entrada i sortida de dades, registra el desenvolupament de la prova.

Número Ciclomàtic: mètrica de software que proporciona una mesura quantitativa de la complexitat lògica d'una funció o programa. Ex: En un graf connectat V(g) = E - N + 1; E és el número de vèrtexs entre nodes, i N és el número de nodes.

Precondició: condició o sentència que ha de ser sempre certa abans de l'execució d'alguna part de codi o d'un cas de prova.

Postcondició: condició o sentència que ha de ser sempre certa després de l'execució d'alguna part de codi o d'un cas de prova.

Script: petit programa no compilat escrit per a un interpretador de comandes o un llenguatge propi.

Stub: simula el comportament d'una part de codi existent o no.

TDD: de l'anglès Test Driven Development, és un model de desenvolupament amb un cicle molt curt. El desenvolupador escriu primer un test, el qual falla, després afegeix el mínim de codi per passar el test. Finalment, aplica els estàndards que se li demanin.

Test Case: unitat organitzativa més petita de l'eina RTRT equivalent a un cas de prova.

Test Class/Test Harness: unitat organitzativa més gran de l'eina RTRT equivalent a un objecte proves o a una classe.

Test Suite: unitat organitzativa mitjana de l'eina RTRT equivalent a un joc de proves.

UML: de l'anglès Unified Modeling Language, és un llenguatge gràfic de modelat de sistemes software. És el més conegut i utilitzat.

Validació: procés d'avaluació del software per a determinar si compleix els requisits especificats.

Verificació: procés d'avaluació del software per a determinar si compleix al final d'una fase de desenvolupament les condicions imposades al principi de la fase.