Treball Final de Grau

EXPERIÈNCIA DE TESTS UNITARIS AMB AUTOMATITZACIÓ VIA VECTORCAST I JENKINS

Grau en Enginyeria de Sistemes TIC

Curs 17/18

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Politècnica Superior d'Enginyeria de Manresa

Autor - Adrián Garcia Gil
Director - Marta Tarrés
Data - 11 d'octubre de 2018
Localitat - Manresa, Barcelona

Agraïments

Diverses persones han col·laborat perquè aquesta experiència fos possible. Primer de tot, em sento totalment agraït amb la meva família per donar-me tot aquest suport durant tota aquesta travessia. Els meus pares són qui m'han donat l'educació i afecte que m'ha fet tenir èxit durant aquesta primera etapa de la meva vida.

Vull donar les gràcies a l'empresa GTD System & Software Engineering per, malgrat la confidencialitat i seguretat rigorosa que els hi demanen als projectes, haver-me permès tractar sobre aquesta temàtica pel projecte. També posar èmfasi amb la incessant ajuda del meu company Joan Garrido, el qual no només ha estat encarant aquests problemes sinó que m'ha donat accés a la creació de Jobs al Jenkins.

Contingut

Agraïments 1

Resum del projecte 4

Abstract 5

Introducció 6

1.1 Unit test 6

1.1.1 Definició 6

1.1.2 Les cinc preguntes sobre un Unit Test 6

1.1.3 Integration tests 7

1.1.4 Entorn de treball 7

1.2 Automatització per un temps prolongat 8

1.2.1 Descripció 8

1.2.2 Jenkins 8

1.3 Objectius i descripció del projecte 9

1.4 Estructura del projecte 10

Estat de l'art 11

2.1 Subversion (SVN) 11

2.2 VectorCAST 12

2.2.1 Introducció 12

2.2.2 Conceptes bàsics 13

2.2.3 Comparació amb altres opcions 14

2.2.4 Facultats del VectorCAST durant els Unit test 16

2.2.5 Conclusió 18

2.3 Jenkins 18

2.3.1 Introducció 18

2.3.2 Conceptes bàsics 19

2.3.3 Comparació amb altres opcions 19

2.4 Llenguatge C++ 20

Descripció del projecte 22

3.1 Introducció 22

3.2 Ús bàsic del VectorCAST 22

3.2.1 Interfície VectorCAST 22

3.2.2 Fitxers bàsics 23

3.2.3 Creació d'un environment 23
3.2.4 Gestió dels Test Case a un mètode 24
3.2.5 Regression scripts 24
3.3 Bones pràctiques 25
3.3.1 Ús de static 25
3.3.2 Fer ús de l'INIT 25
3.3.3 Donar un cop d'ull a les Unstubbed Functions 26
3.3.4 Redirigir el standard output 26
3.3.5 Fer ús del Debugger 26
3.3.6 Localitzar threads en els tests 28
3.3.7 Consultar els fitxers .tst anteriors 28
3.4 Resolució de problemes 28
3.4.1 Desaparició de l'informe després d'una execució 28
3.4.2 Retorn de diferents valors als stubs 29
3.4.3 Detecció d'una classe dins un namespace 29
3.4.4 Redirecció del standard input 29
3.4.5 Problema de creació d'un objecte concret a partir d'un abstracte 29
3.4.6 Problemes amb els shared_ptr 30
3.4.7 Impossibilitat de stubeig de funcions Template 31
3.4.8 Stubeig de classes Template 31
3.4.9 UUT amb dependències a altres classes 31
3.4 Configuració del Jenkins 33
3.4.1 Creació d'un projecte Manage 33
3.4.2 Creació d'un VectorCAST Job a Jenkins 33
3.4.3 Visualització dels resultats 37
3.4.4 Problemes trobats 38
Desplegament del projecte 40
Resultats de l'experiència al projecte 41
Conclusions 44
Bibliografía 45

Resum del projecte

Quan es desenvolupa programari, a part de la tasca bàsica d'escriure codi, la detecció i prevenció d'errors és un punt clau si volem que l'eina que es crea funcioni com és degut. Aquesta meta l'aconseguim mitjançant el test del codi resultant de cada fase clau del projecte.

Avui en dia hi ha eines que ens proveeixen del necessari per cobrir aquesta necessitat: JUnit, Unittest, CppUnit… Tot depèn del llenguatge en què es programi i les preferències de la plantilla. Tot i això, cap de les utilitzades per la majoria del públic té característiques diferencials que els separin de la resta.

És per això que he decidit aprofundir en una eina anomenada VectorCAST, que és revolucionària en certs aspectes:

- Introdueix el test i el seguiment de codi per interfície gràfica.
- Ofereix solucions que automatitzen parts del procés i ajuden a cobrir tots els casos dotant de flexibilitat per certs canvis en el programa.
- Suport per Test Driven Development (TDD), per tal de poder crear els tests abans de tenir codi per poder anar comparant cada mòdul de codi que sorgeixi.

Malgrat tot el que ofereix, s'ha d'adaptar amb altres parts del projecte. Aquestes poden ser el control de versions per gestionar cada actualització que rep el codi font (.hpp, .inl i .cpp), documentar tots els resultats de cada passada dels tests (els valors esperats es compleixen, nombre de condicions i línies dins d'una funció cobertes, complexitat ciclomàtica, on hi ha errors, entre d'altres) que es fan més sovint que els canvis en el codi, instal·lació…

Per això, parlaré del programari d'integració contínua Jenkins que ajuda al següent:

- Automatitza l'execució dels tests (ja sigui disparant una actualització d'una eina de control de versions, cada cert temps, etc).
- Organitza i mostra la documentació en qualsevol plataforma que tingui accés a un cercador d'internet.
- S'integra amb les altres facetes del projecte de software: el control de versions, els tests i el procés d'instal·lació a màquines remotes o virtuals.
- Permet l'accés a la documentació creada i la visualització de resultats si s'instal·la finalment a tothom qui tingui el permís pertinent.

Abstract

While developing software, it is not only about writing code, the detection and prevention of errors is a key point if we desire that the tool that we are creating respond properly. This goal is reached through the test of the resulting modules of each main stage in the project.

Nowadays, there are solutions that provide the necessary services to cover that need: JUnit, Unittest, CppUnit… The choice depends on the language we program and the preferences of the team. However, any of the most used by the major part of the public has breaking features that make them outstand of the crowd.

That is why I have decided to give a deep approach of a platform called VectorCAST, which is revolutionary in certain aspects:

- Introduces the test process and the tracking of the code with graphical interface.
- Offers unique traits that help to automate part of the process and assist to cover all the possible cases supplying flexibility in concrete changes of the program.
- Support for Test Driven Development (TDD), in order to create the tests before the code is done.

Despite all that offers, it is necessary to adapt it to other parts of the project. In the possible options we have source code management tools in order to control each update of the source code (.hpp, .inl and .cpp), document every result of each run with the tests (expected values, number of statements and branches covered, cyclomatic complexity, bug detection, and more) which are made more often than the changes in the code, installation…

That is the reason why I will talk about the continuous integration software Jenkins that helps with the following:

- Automate the execution of the tests (either with a change in the source code control tool, within a period of time, etc).
- Organises and displays the documentation in any platform with access to a browser.
- Integrates several fields of the development: the version control, the testing and the installation in remote or virtual machines.
- Allows the whole team to see the documentation and the results of the installation if they have accreditation.

Introducció

L'addició de tests a un projecte aporta grans beneficis a curt i a llarg termini. Entre aquests es troben l'agilització a l'hora de modificar o afegir codi a un d'existent, millora la qualitat del codi gràcies al fet que detecta fàcilment els bugs per la seva simplicitat i facilitat d'executar-los periòdicament i proporciona documentació al desenvolupador. Tot això dóna com a resultat una reducció de costos del projecte per no haver de dedicar tant temps a tasques que ateses abans acaben sent molt més fàcils que si s'encaren quan ja està el projecte molt avançat.

1.1 Unit test

Si hi ha cap terme que ressoni en qualsevol projecte de software pel que fa a manteniment del codi d'una empresa és el testing. Però, realment ens aporta avantatges? Raonem-ho més endavant. Tot i això, aquest es pot pensar amb diferents perspectives depenent del que s'està valorant del codi. Aquí es descriuran els punts principals del unit testing.

1.1.1 Definició

Segons en Roy Osherove, al seu llibre The art of Unit Testing [1] (Osherove, 2013), un unit test és un tros de codi (normalment un mètode) que invoca a una altra peça de codi i comprova unes certes suposicions més tard. Si les suposicions resulten ser incorrectes, l'unit test ha fallat.

1.1.2 Les cinc preguntes sobre un Unit Test

Tenint en compte que la definició no marca clarament les propietats bàsiques s'ha de trobar un patró per distingir les propietats característiques d'un unit test.

A partir de la qüestió "com puc saber que estic fent un unit test adientment?" sorgeixen les cinc preguntes que deixen clar si s'està seguint el camí correcte [1] (Osherove, 2013):

1. Puc executar els tests que vaig escriure fa un temps? Un dels trets més característics és que han de poder repetir-se al cap d'un període de temps, els resultats segueixen sent rellevants i consistent entre execucions (que no canvien els resultats entre dos cicles del mateix test).
2. Pot qualsevol membre del meu equip executar el test sense dificultat i rebre uns resultats? Els tests han de ser suficientment entenedors com perquè qualsevol persona pugui activar-los i rebre un feedback comprensible. D'aquesta manera, si hi ha qualsevol fallada als tests, és fàcil detectar on es troba l'error.
3. Puc executar els tests que he escrit en no més d'uns minuts? Un dels adjectius que defineixen els unit test és que són fàcils d'implementar. Això ve donat en part també perquè estan fets per ser executats en un temps curt.

4. Puc executar tots els tests que he escrit amb només polsar un botó? Els tests han d'estar prou automatitzats perquè no hi hagi un llarg procediment de configuració per activar els tests que es volen dur a terme en un moment donat.

5. Puc escriure un test bàsic en no més d'uns minuts? A part del que s'ha dit al punt 3, hi ha altres punts a favor: la fàcil detecció dels resultats esperats a partir del mètode que s'està analitzant (tenint el control de tota la UUT), un test és independent d'un altre test.

1.1.3 Integration tests

Si no s'ha pogut respondre afirmativament les preguntes anteriors, no és que s'hagi fet un test erròniament, sinó que en comptes d'un unit test s'estava fent un tipus també molt important, un integration test.

Aquests poden ser descrits per no tenir control total de la unitat de test, utilitzar diverses dependències reals (com la data actual, bases de dades…). Això comporta el següent:

- Si utilitza un paràmetre real com el temps, cada cop que s'executi el resultat esperat serà diferent. Deixa de ser consistent.
- Si tracta una base de dades, el test deixa de treballar només en memòria, per tant, els canvis realitzats per la prova són més difícils d'esborrar que treballant amb dades falses fetes pel test. També executant-se fora de memòria fa que tardi més a acabar.
- Amb aquesta manera de procedir, es cau en el risc de comprovar massa punts alhora. Això comporta a part d'un increment de complexitat a l'hora d'escriure tests, un augment en el temps d'execució i de la dificultat de trobar bugs al codi.

1.1.4 Entorn de treball

El nombre de plataformes disponibles per realitzar unit tests és enorme. Com podem escollir la més adient pel nostre propòsit? Això depèn de diversos factors. Segons de la rigorositat de cobertura que es demani (percentatge de línies i condicions cobertes, nombre de resultats esperats verificats, etc.), si el sistema a analitzar funciona a temps real o no, el llenguatge en què estigui escrita l'aplicació… Determinaran l'elecció d'un software concret o un altre.

Per contextualitzar la situació ara s'explicarà el projecte amb el qual he adquirit experiència en el tema. Treballant per l'empresa GTD, en un projecte de l'àmbit espacial, he estat testejant els mòduls encarregats de la comunicació via Ethernet entre les diverses parts del coet Ariane 6 i la taula de comandaments.

Aquest tipus de software està catalogat segons l'estàndard DO-178B [2] (2017) (Software Considerations in Airborne Systems and Equipment Certification) com a nivell B de criticitat (sent el nivell A el més perillós). El nivell B engloba tot aquell software amb el qual qualsevol fallida del programa té un impacte negatiu a la seguretat, el rendiment o capacitat de la tripulació d'operar l'aeronau a causa d'un increment en l'estrès a causa de lesions serioses o fatals entre els passatgers. Per tant, se'ns requereix un percentatge de statements o cobertura de les línies de codi d'un 100% i un 90% dels branches o condicions que es poden donar dins dels mètodes o funcions implicats.

Pel que fa a l'organització d'equip hem treballat 3 persones dins d'aquesta tasca, cobrint aproximadament 2.000 línies de codi en C++. A part de la comunicació entre nosaltres via

correu electrònic, hi havia també la necessitat de parlar amb 3 desenvolupadors encarregats dels mòduls coberts, el TTE_COMMON i el TTE_GATEWAY. Els errors o possibles millores del codi font eren transmesos a través d'una plataforma de tasques anomenada Redmine, un portal web propi.

Tenint en compte que les comunicacions han de ser a temps real, hi ha un alt nivell de complexitat en el codi, fent servir dissenys software complexes, llibreries molt concretes i utilitzant regles de codificació molt estrictes.

VectorCAST és conegut per proveir eines i serveis pel desenvolupament de sistemes electrònics que requereixen una qualitat òptima on el procés pugui ser automatitzat. Aquests casos es troben als sectors aeroespacials, d'automoció, medicina, entre altres.

Aquest, se centra en els llenguatges de programació Ada i C++. Entre les seves facultats destaquen el suport per temps real i estar fets a mida per sistemes encastats i els seus plugins d'automatització.

Atès que analitzo un software crític escrit en C++ i que VectorCAST treballa amb aquestes certificacions és un gran exponent per aquesta tasca.

1.2 Automatització per un temps prolongat

1.2.1 Descripció

Com s'ha deixat clar anteriorment, els unit test han de ser fàcils de tornar a executar per tal de veure ràpidament els errors entre canvis de versions.

Una possible solució seria que cada dia abans de començar a desenvolupar cada persona de l'equip executés tots els tests. Ara bé, si hi ha un gran nombre de classes, el procés ja no és tan trivial com perquè tardi només uns minuts. D'una altra banda, el que valdria per tots els membres de l'equip realitzat un sol cop, ho estarien repetint cadascun en el seu ordinador. Queda clar que no és una bona manera de procedir.

Una millor manera d'encarar aquest problema seria automatitzar el procés. Ja sigui cada cert període de temps, s'executin tots els unit test o cada cop que hi hagi un canvi en el codi (detectat per control de versions per exemple). Aquí és on entra en joc el software d'automatització Jenkins.

1.2.2 Jenkins

El Jenkins és un software escrit en Java que pot ser instal·lat fàcilment a tota mena de sistemes operatius que permetin fer ús de navegador d'internet. Tanmateix, la configuració del Jenkins no ha de ser ni per només un ordinador (localhost) ni per una xarxa local, ja que limitaria el nombre de dispositius que hi poden accedir. Cal afegir que la interacció amb la seva senzilla interfície des de qualsevol ordinador a un projecte amb les credencials adequades alleugera molt la feina.

Amb la seva ajuda, no només s'automatitza el procés de proves del codi, sinó que també es detecten bugs, es pot comprovar la cobertura del codi amb els unit test, analitzar mètriques i regles per la seva qualitat, monitorar el procés de desenvolupament i generar documentació tant per l'empresa com pel client.

Tal com es veurà més endavant, es tracta d'un software molt personalitzable i escalable per tal d'obtenir una solució molt més a mida. En el cas que es produeixi un programa o app per diverses plataformes, sistemes operatius o inclús versions del mateix es pot dividir cada cicle d'integració.

A tota aquesta manera d'actuar per obtenir els beneficis descrits anteriorment s'anomena integració continua i és molt habitual quan es tracta amb software i sistemes informàtics. A més, pot ser utilitzat amb altres bones pràctiques com l'agile development, d'entre altres.

1.3 Objectius i descripció del projecte

Aquest projecte té tres objectius principals:

- Aprendre i entendre una quantitat considerable d'eines relacionades amb la integració de sistemes i test, una temàtica en la qual no estic familiaritzat.
- Conèixer els aspectes a tenir en compte d'un llenguatge relativament nou per mi, el C++. No és totalment desconegut, ja que comparteix similituds molt grans amb el C.
- Com a estudiant i futur enginyer, intentar combinar les dues disciplines en un projecte real de l'empresa GTD.

Voldria reflectir en aquest document el que he extret d'aquesta vivència, que espero que sigui d'ajuda en treballs propers tant per mi com per qualsevol lector d'aquest.

Per l'assoliment dels meus objectius he utilitzat VectorCAST i Jenkins, un software propietari "tancat" per segons quines tasques en contrast amb un de codi obert amb infinitat de plugins que aporten molta flexibilitat. Per tant, una gran part del meu treball ha sigut entendre els límits i capacitats d'aquestes eines. No només això, adquirir el criteri per desenvolupar un entorn capaç d'interconnectar els dos programes.

El procés d'obtenció de dades comença amb la pujada de codi per part de l'equip de desenvolupadors de codi al control de versions SVN. En el cas que es tracti d'una classe nova es crea el nou entorn de VectorCAST i un cop coberta (o fins a un nivell acceptable) es puja al SVN perquè el Jenkins el mostri a la documentació i les gràfiques. Si en canvi ha sigut una modificació de codi testejat, veiem que la cobertura ha baixat a la documentació del Jenkins i prosseguim a pujar-la al VectorCAST. El procediment es pot veure a la Figura 1.1.

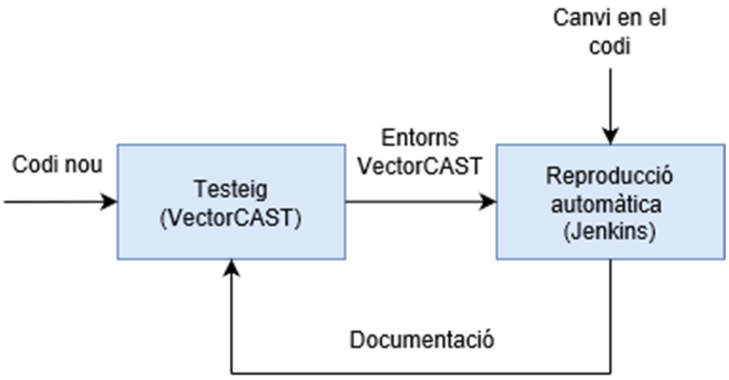


Figura 1.1 Cicle dels tests[1]

Finalment, remarcar com a aspecte positiu el fet de combinar dos softwares d'ideologies tan diferents m'ha fet intentar adaptar a dues maneres de pensar totalment diferents entre elles. Contràriament, treballar amb aquestes dues grans solucions m'ha plantejat les següents adversitats:

- Treballar amb una quantitat immensa d'informació i amb manca d'exemples concrets no tan ideals com els teòrics.
- La coordinació amb altres companys que s'encarregaven de la mateixa feina que jo i els que es dedicaven a altres (escriure codi, etc.) juntament amb l'inconvenient d'haver de descobrir el llenguatge C++, que aporta diferents qualitats mai vistes al C.

1.4 Estructura del projecte

Per tal de distribuir el millor possible aquest projecte he decidit que he de procedir de la següent manera:

En el capítol 2 a part d'una secció de conceptes bàsics, s'introduirà l'estat de l'art de les tecnologies tractades: El control de versions Subversion (SVN), el VectorCast, el Jenkins i finalment el llenguatge C++.

Al capítol 3 es descriu el projecte on es podran veure casos concrets amb els serveis estudiats en profunditat.

El capítol 4 explicarà l'entorn on es desplega el projecte, que per motius de confidencialitat i privacitat no serà en gran detall.

Per acabar, concloent el treball, el capítol 5 discutiré els resultats de les proves fetes i al capítol 6 proveirà les conclusions finals.

[1] Imatge pròpia

Estat de l'art

Aquest capítol descriu la tecnologia utilitzada durant aquest projecte. Posaré èmfasi en el VectorCAST, que és el nucli del projecte, i acabaré amb una explicació del Jenkins.

2.1 Subversion (SVN)

Per tal de tenir el codi accessible per tot l'equip, mantenir un control de cada versió operativa amb la qual operem i poder desfer errors humans utilitzem un software especialitzat com és Subversion.

Aquesta selecció no ha sigut poc meditada. Tenint en compte que no hem de tenir el codi exposat a tercers com GitHub ni s'ha de pagar cap mena de llicència el fa un gran exponent en aquest cas. A més, el fet de funcionar per línia de comandes el fa senzill i molt adaptable si treballem només amb un terminal.

Per iniciar un projecte al SVN és tan fàcil com descarregar-se una còpia de treball (checkout) o pujant els fitxers o directoris propis amb un import. En tot cas es necessiten les credencials per accedir al repositori i la direcció o localització d'aquest.

Un cop ja s'està treballant en un projecte amb Subversion l'ordre correcte de treball amb aquesta eina és el següent [3] (Collins-Susman, et al., 2006):

1. Actualitzar la còpia de treball. Utilitzant un update es pot sincronitzar la còpia de treball amb l'última versió del repositori.
2. Fer els canvis propis. Aquí s'inclouen els canvis en l'àmbit de l'arxiu i de l'arbre. No és necessari notificar al Subversion per fer un canvi dins d'un fitxer (renovar el contingut d'un arxiu amb un editor per exemple), en canvi, parlant sobre l'arbre, on s'afegeixen o eliminen fitxers entre d'altres, es "concerta" el canvi amb el SVN mitjançant una ordre específica. Els canvis tant d'aquest últim com els anomenats anteriorment no es veuran reflectits al repositori fins que no es publiquin amb un commit.
3. Revisar les modificacions fetes. Abans de reflectir els canvis fets al repositori és convenient mirar quins fitxers o directoris han sigut modificats. Amb la comanda status es pot veure si estan o no sota control de versions, si estan per afegir, s'han esborrat, hi ha algun canvi o si hi ha conflicte entre versions. Si es vol fer amb més detall, es pot utilitzar l'ordre diff per veure quines línies s'han esborrat i quines s'han afegit.
4. Arreglar els possibles errors. Una opció molt útil d'aquesta solució és tornar a la versió original després d'haver-la modificat. Aquest recurs és accessible gràcies al revert.
5. Resolució de possibles conflictes. Si a l'hora d'actualitzar la nostra còpia de treball trobem un conflicte entre versions, es notificarà i apareixerà un seguit d'opcions per solucionar-ho, ja sigui amb la part pròpia del conflicte, la d'un altre usuari...
6. Pujar els canvis duts a terme. Aquesta part és bastant directa si tot s'ha fet com és degut. S'accionen els canvis amb la comanda commit, es poden afegir missatges i especificar quins arxius vols incloure en el canvi de versió.

Com a norma de bona pràctica mai s'envia cap arxiu que no compili, no funcioni correctament o estigui malmès a l'SVN, en el cas de voler-ho automatitzar, el Jenkins proporciona el servei de primer passar-li les proves pertinents i si tot és correcte, es puja al control de versions. En el cas de dubte de canvis recents o qui els ha fet és útil recórrer a l'ordre log.

2.2 VectorCAST

> "Eighty percent of the errors are found in twenty percent of a project's classes or routines." [4] (McConell, 2004)
>
> (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).

> "Ninety-five percent are caused by programmers, two percent by systems software (i.e.; compiler and OS), two percent by some other software, and one percent by the hardware." [4] (McConell, 2004)
>
> (McConnell, Steve. Code Complete. Redmond: Microsoft Press, 2004)

De les dues frases anteriors podem extreure dos punts molt importants en el món del testing:

- L'àmbit o abast de la majoria d'errors és bastant limitat.
- Els errors més comuns provenen del programador.

A partir d'aquestes dues premisses tan importants el VectorCAST intenta centrar-se a posar-hi remei com descriuré en els següents punts.

2.2.1 Introducció

La plataforma d'automatització de tests VectorCAST permet fer tests dinàmics a través del cicle de desenvolupament amb suport dels llenguatges C, C++ i Ada [5] (VectorCAST, 2017). La versió escollida per aquest projecte és la VectorCAST 6.4. Les organitzacions de desenvolupament de software que necessiten solucionar problemes complexes de qualitat l'utilitzen per proveir consistentment resultats manejables i repetibles que redueixen el temps i el cost dedicat al projecte proporcionant un codi segur, fiable i que satisfà els requeriments inicials. Aquesta versàtil solució és emprada per tests d'aplicacions en indústries variant des de l'aeroespacial i defensa fins a la mèdica.

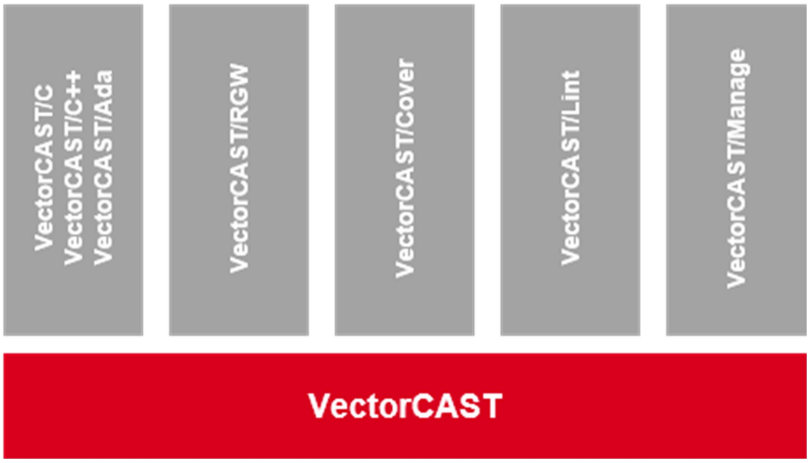


Figura 2.1 Divisions de VectorCAST[2]

Com es pot veure a la Figura 2.1 entre els serveis que ofereix està l'automatització de tests amb la secció VectorCAST/(C,C++,Ada), una bona traçabilitat dels requeriments amb el mòdul de RGW (Requirements Gateway) que serveix per exportar/importar els requisits d'altres tipus de fitxer(.doc, .txt, .csv...), la cobertura i generació d'informes amb l'apartat Cover, l'anàlisi estàtic i reforçament d'estàndards de codificació amb Lint i les eines d'integració contínua gràcies al Manage.

Una opció que no han deixat de banda és la possibilitat de fer Test Driven Development. Aquest terme es refereix a l'elaboració de codi de manera que a partir del disseny i els requeriments es produeixen els tests amb els valors a comparar escaients i cada cop que sorgeix una part de software funcional es prova contra els tests successivament fins que tots passin. Aquest esforç és només possible perquè et permet crear l'entorn de treball sense tenir cap classe escrita.

Segons la documentació de VectorCAST s'ha de tenir uns mínims de sistema operatiu:

- Pels usuaris d'entorn Windows és necessari Windows 7 o superior i en el cas de servidors Windows server 2003 com a mínim.
- Pels usuaris d'entorn Linux qualsevol sistema operatiu amb la llibreria GNU C (glibc) amb la versió 2.5 o posterior és capaç d'utilitzar-lo.
- Per Solaris 8 o posterior (tot i que la versió 2018 de VectorCAST no suporta Solaris).

Per acabar, cal afegir que permet obtenir certificats pel codi i ofereix la possibilitat de connectar amb altres softwares dedicats a aquest camp com Simulink, Jenkins, Simics, etc.

2.2.2 Conceptes bàsics

Unitat - Un fitxer de codi font (*.c/*.cpp) o un paquet d'Ada.
Unit Under Test (UUT) - Una unitat que conté funcions que seran testejades.
Statement - línia de codi font que ha de ser coberta.
Branch - cada valor que pot prendre una condició expressada en un statement (variable bool, switch, if, while, for...).

[2] Imatge pròpia

Subprogrames visibles - Subprogrames o funcions de la UUT que són testejables.
Unitat dependent - Una unitat que conté funcions que són utilitzades per la UUT.
Stubs - Trossos de codi font (normalment crides a classes i mètodes) que són substituïts per la resposta volguda per aïllar la unitat al màxim.
Test Driver - Programa principal que executa els programes visibles de la UUT.
Test Harness - Programa executable crea quan el Test Driver, les Unitats dependents, les Stubs i les UUTs són enllaçats.
Test case - Col·lecció de dades d'entrada per estimular la UUT i dades esperades per comparar amb la sortida de la UUT.
Whitebox - El Test Harness té visibilitat a l'estructura interna de la UUT.
Instrumentació - Codi afegit per VectorCAST al codi font propi per portar el seguiment dels statements i branches que han sigut executats.
Breakpoint - Marca que se li posa a un mètode per tal d'aturar el programa en un punt determinat i examinar algun comportament anòmal, utilitzat per debugar.
User Code - Codi que s'afegeix al test per crear variables, objectes i d'altres que no siguin de tipus bàsic (int, float, ...) que poden ser entrats per interfície gràfica.

2.2.3 Comparació amb altres opcions

Per tenir una comparació fidedigna he considerat no només comparar-lo amb els frameworks dedicats a testejar codi sinó contrastar-lo amb contrincants que no es configuren majoritàriament per interfície gràfica, s'ajusten per codi o línies de comandes.

Les tres opcions són plataformes madures d'uns 20 anys d'experiència en els que han pogut evolucionar, elaborar suficient documentació i guanyar adeptes. El software tingut en compte és:

- VectorCAST
- CppUnit
- Boost.Test

Per poder comparar-los he intentat buscar uns trets que puguin ser qualitats de pes per decantar-se per ells com una bona solució. Els que més adients he trobat han sigut els que apareixen en un dels articles contrastats [6] (Noel, 2010) [7] (Main, 2010):

- Treball mínim per afegir tests
- Fàcil de modificar i portar a altres plataformes
- Fàcil de modificar i portar a altres plataformes
- Suport de creació abans del test d'objectes/variables
- Manipulació adient d'excepcions i parades
- Bones opcions de comparació de resultats
- Capacitat de variar els medis on mostrar els resultats
- Capacitat de variar els medis on mostrar els resultats
- Opció per executar tests per grups (suites)

En el cas de VectorCAST ens trobem amb el següent:

- El treball que s'ha de fer és mínim pel fet que només has d'accedir a la casella on es troba la variable, mètode o objecte en qüestió i omplir-la amb el valor volgut.

- No només no oposa problemes amb diferents plataformes, sinó que es pot classificar dins del projecte distingint compiladors diferents i elaborar els scripts de regressió tant per un entorn Windows com per un Linux.
- Pots crear un objecte base que s'utilitzi per a tots els tests de la UUT, fer-ne un de concret per cada test o combinar les dues a la vegada.
- Les excepcions i aturades inesperades surten al report després d'executar el test, però si es vol informació més concreta, sempre es pot recórrer al debugger.
- Pel que fa a la comparació de resultats tots es veuen al report també. Amb l'excepció de l'User Code que ha fallat que hem de veure per debugger.
- En aquest cas és una mica restringit, però tens l'opció de mostrar el report en html, en text pla, redirigir el stdout al report, etc.
- Tot per interfície gràfica és molt fàcil i intuïtiu d'agrupar en seccions del projecte per carpetes, tipus de compiladors, etc.

Per l'entorn més utilitzat pels tests unitaris de C++, el CppUnit, tenim:

- S'ha d'escriure bastant codi per configurar tot l'entorn del test, cosa que repercuteix a altres aspectes com ara es veurà.
- Sí que es pot portar a altres plataformes, tot i això, el problema anterior i les dependències amb llibreries molt específiques dificulten la feina.
- Permet crear els objectes abans de cada test.
- Aconsegueix mostrar les excepcions correctament i dóna opció de personalitzar com es fa.
- Té una petita manca respecte a les afirmacions de menor que i major que. No obstant això, és bastant complet.
- Gràcies als seus "outputters" i "listeners" podem canviar entre diferents sortides i inclús ser notificats.
- Admet l'ús d'agrupacions de tests.

I amb l'últim, el Boost.Test es pot extreure:

- S'ha d'escriure molt poc codi per cada test, això sí, la quantitat de codi a escriure augmenta considerablement quan es volen agrupar.
- Té el mateix problema que l'anterior, suposa feina portar el projecte però es pot fer.
- No funciona amb el típic esquema de Setup/Teardown, en aquest cas ho fa com el VectorCAST amb Constructors/Destructors però també admet la creació prèvia al test d'objectes, etc.
- És el millor exponent en aquest aspecte. No només captura les excepcions sinó que també mostra informació sobre aquestes.
- Conté tots els mètodes de comparació necessaris que es puguin demanar amb la inclusió de mirar si ha sortit una excepció.
- Segons la documentació pròpia de la plataforma ho permet mostrar en XML i de descripció en línia de comandes.
- Permet utilitzar-les, però suposa un gran esforç d'escriptura del codi.

Com es pot veure, la diferència més gran recau en dos punts molt importants que no són els que s'analitzen normalment: l'ús d'interfície gràfica o codi per interactuar amb ells i el fet de ser un software privatiu o de codi obert.

La primera característica pot ser més qüestió de gustos, al que un s'acostumi. Tot i això, el fet d'haver de pagar una llicència, que parteix dels 10.000 € depenent del nombre de llicències i els components que es contractin, una quantitat no menyspreable, pot fer

retrocedir a alguns. Per altra banda, perdre flexibilitat davant del codi lliure pot ser decisiu en segons quins casos. També cal tenir en compte que ho compensa amb les eines d'automatització que es mostraran a continuació.

2.2.4 Facultats del VectorCAST durant els Unit test

Per tal de ser precís i resistir el màxim als canvis el VectorCAST ens proporciona un seguit de facilitats en segons quins camps de l'unit testing. S'anomenaran les més significatives [8] (VectorCAST, 2018):

Basis paths
Per tal de tenir el 100% de cobertura del codi, no només s'han de cobrir totes les línies de codi o statements, sinó que també cada branca o branch on hi ha condicions (if, for, etc).

Si es clica al botó dret i accedim a Insert basis paths es generarà automàticament el nombre exacte de tests per la funció en qüestió. En el cas que hi hagi algun objecte o variable que no sàpiga inicialitzar sol ho indicarà abans de la seva creació.

Aquests basis paths es creen basats en la complexitat ciclomàtica del mètode testejat, en el cas de donar >10 (més de 10 basis paths) el codi deixa de ser sostenible i és recomanable la reconfecció d'aquest.

Límits mitjançant tags
Com les variables poden canviar de tipus, i amb això, el rang que poden representar, el Vector cast presenta una opció que permet posar la variable d'entrada o de sortida amb una etiqueta (<<MAX>>,<<MIN>>). Així, si el programador canvia, per exemple, d'unsigned int a int, el valor mínim passarà de -X a 0 sense haver de canviar els tests. Aquestes es poden introduir via el botó dret del ratolí o escrivint-les en text pla dins la casella.

Aquestes també poden utilitzar-se per veure el comportament del programa quan es passa el valor per sobre o per sota del límit màxim i mínim (<<MAX+1>>,<<MIN-1>>). Un bon exemple seria la Figura 2.2.

Figura 2.2. Ús d'etiquetes o tags[3]

Variable tree
Quan una variable deixa de ser d'un tipus bàsic de C++ (és un objecte d'una llibreria, o en algun cas un punter d'una llibreria a un objecte propi) s'ha d'escriure com a User Code, ja que la interfície no pot oferir les opcions per incloure el valor desitjat. Per objectes i variables del nostre projecte que el VectorCAST ha trobat podem arrossegar cap a l'User Code des de la finestra Variable tree sense haver de pensar quin nom tenen.

[3] Imatge pròpia

Coverage i Debugger

Per veure el curs de l'execució del nostre test hem de seleccionar l'opció de Coverage. Es pot veure marcat en verd (la part coberta), en groc (el branch que només està cobert amb una de les dues possibilitats, true o false) i vermell (la part que queda per cobrir) al codi. Un cop executats es pot marcar i desmarcar, es pot veure el seu funcionament a les Figures 2.3 i 2.4.

Figura 2.3. Marcatge dels Basis Paths que es vol mostrar[4]

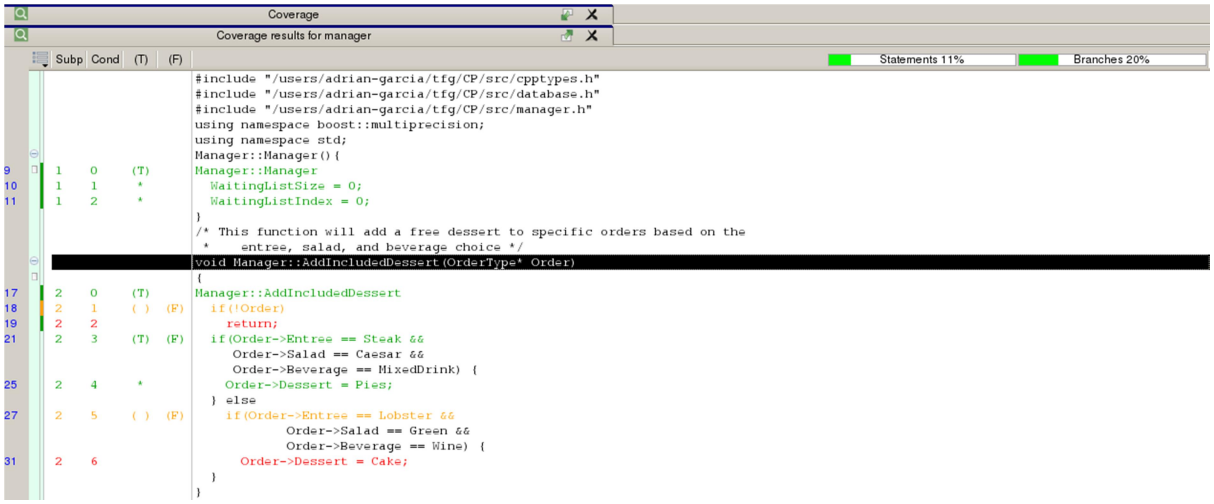


Figura 2.4. Vista del curs del programa i el percentatge cobert[5]

En el cas que l'execució acabi inesperadament abans d'acabar totes les línies per on ha de passar, es pot utilitzar el Debugger. Consta d'un terminal, amb el que podem veure l'error en concret, posar breakpoint per tal de recórrer les línies d'execució una per una, introduir-se en la crida a una funció dins del mètode, imprimir variables, etc.

Regression Scripts

Per tal d'automatitzar el procés de reproduir els tests o simplement repetir-los quan es vulgui, VectorCAST ofereix una eina anomenada Regression Scripts. Amb tres fitxers bàsics es pot recrear l'environment i executar-lo just després obtenint la seva respectiva documentació. Això a més, ens permet no haver de conservar el directori amb el harness code de cada classe que cobrim i d'altres fitxers.

Stubs

Una bona manera d'aïllar el mètode que estem sotmetent a un test és utilitzar stubs per crides a altres funcions. Això permet que sigui més resistent a canvis, ja que reemplacem la

[4] Imatge pròpia
[5] Imatge pròpia

crida d'una funció per la resposta volguda per aquest test en concret. Això sí, els mètodes substituïts han d'estar correctament coberts per tal d'evitar comportaments erràtics.

2.2.5 Conclusió

A part que és palpable el nivell d'automatització que s'obté amb el programa de VectorCAST està clar que també aborda els dos problemes amb els quals s'ha començat aquest punt:

Concentrem els errors fàcilment amb els stubs que aïllen el màxim possible els mètodes, el suport pel Test Driven Development ens ajuda a poder cobrir fallades comparant cada peça de codi que desenvolupem i els basis path ens permeten no perdre de vista cap possible recorregut del codi.

Aporta flexibilitat als canvis i control d'errors de tots els tipus amb els tags i permet la reproducció dels casos investigats de manera senzilla amb l'automatització dels mateixos amb el Jenkins.

2.3 Jenkins

2.3.1 Introducció

El Jenkins és un programa basat en el llenguatge de programació Java, que constitueix un cop instal·lat un servidor d'automatització molt fàcil de configurar gràcies a la seva simple interfície. Aquest, pot ser utilitzat com a servidor d'integració contínua com en aquest projecte o com un desplegador de codi automàtic. Ofereix una flexibilitat increïble, tant per la seva instal·lació en quasi qualsevol sistema operatiu com per la possibilitat que ens aporta amb els plugins que ofereix. Inclús un mateix es pot fer plugins a mida.

En ser un software multiplataforma només es demana uns requisits mínims de hardware i el següent software (la versió utilitzada pel projecte és la 2.124) [9] (Jenkins, 2018):

- Un mínim de 256 MB de RAM i 1 GB d'espai de disc dur (encara que per un equip petit aconsellen 1 GB o més de RAM i almenys 50 GB d'emmagatzematge).
- Requereix Java (tant per 32 com per 64 bits i estan treballant per oferir servei per Java 10 i 11).
- Pel tema dels cercadors els que tenen les últimes versions abans sempre són Google Chrome i Mozilla Firefox però també hi ha certes versions per Safari i Internet Explorer.

Un dels trets més importants és la seva estructura, que el fa molt escalable pel balanç de càrrega, això comporta que no se satura tant un sol recurs, millora el temps d'espera, entre d'altres. Aquest fet es veu a través del seu mètode master-slave que permet utilitzar diverses màquines (remotes o virtuals) per fer diversos jobs concurrentment. Presenta un esquema molt fàcil de descriure tant programant com esquemàticament via pipelines (vegeu Figura 2.5). Les màquines virtuals també ens habiliten a desplegar un software per diferents plataformes.

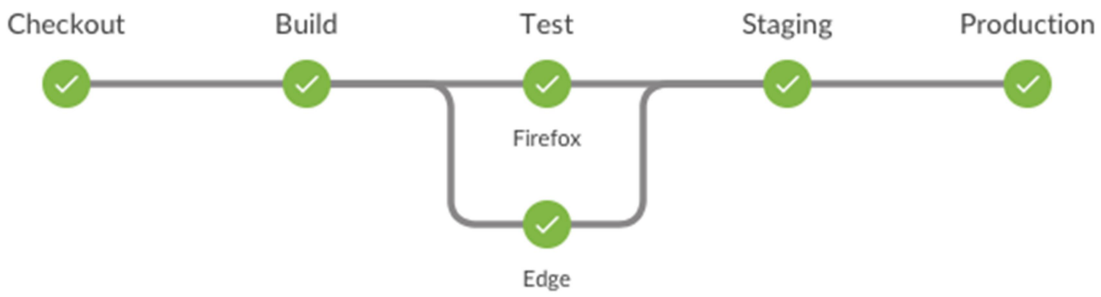


Figura 2.5. Pipeline de Jenkins[6]

2.3.2 Conceptes bàsics

Plugin - Petita "peça" de software que es pot afegir o treure d'un programa molt més gran per aconseguir funcionalitats addicionals.
Job - Procés del Jenkins on s'executa qualsevol acció amb algun dels seus plugins.
Pipelines - Entorn on s'enllacen diversos jobs dins del mateix projecte.
Executors - Són els encarregats d'executar els Jobs, com més executors li proveïm al Jenkins més Jobs poden tenir lloc concurrentment (normalment es posen el nre. de nuclis que tenim al processador per optimitzar la tasca).

2.3.3 Comparació amb altres opcions

Com que comparar el Jenkins amb el servidor d'integració contínua Hudson no té molt sentit, ja que Hudson és la versió privativa de Jenkins quan Oracle el va comprar ens centrarem amb un altre gran exponent com TeamCity.

Es tracta d'una solució que no és de software lliure, pel que si ens excedim dels 100 Projectes i 3 Build Agents (màquines amb les quals repartir el treball) hauríem de pagar una de les llicències que l'empresa JetBrains ofereix.

Dins de les característiques que ofereix hi podem trobar[10] (JetBrains, 2018):

- Seguiment de les accions de cada usuari en tots els projectes agrupant els usuaris amb diferents tasques i permisos.
- Capacitat de guardar una còpia de seguretat tant de les configuracions del servidor com dels executables i la documentació resultant dels diferents cicles que s'han fet d'un projecte.
- Afegir l'opció de testejar primer el codi abans de pujar-lo a control de versions, així, ens adonem de l'error molt abans i no tenim codi erroni al control de versions. A la vegada, també ajuda utilitzant codi de diferents fonts simultàniament (SVN, GitHub...).
- Ofereix més de 300 plugins per tal d'incloure els procediments (tests unitaris, anàlisi estàtica de codi, desplegament...) que es vulgui amb les plataformes que tinguin compatibilitat gràcies a ells. Cal afegir que es pot crear un plugin propi programat en Java gràcies a l'OpenAPI.
- Es poden dur a terme tasques entre diverses màquines físiques o virtuals per distribuir la càrrega de treball.

[6] https://liatrio.com

- Té una gran capacitat de configuració. Aquesta inclou jerarquia de projectes, aprofitar configuracions d'altres jobs (meta-runners) i prevenció de recursos compartits.

Com es pot veure, es tracta d'un servidor molt extens i que pot variar molt segons quins complements s'utilitzin. Tot i això, no he sabut veure cap tret diferencial que pogués decantar la balança de banda del software TeamCity en el tipus de projecte que vull desenvolupar (totes aquestes eines anomenades anteriorment també les proveeix el Jenkins). És més, a part que està limitat si no es paga una llicència, no té integració amb el software de test que vull utilitzar, el VectorCAST.

2.4 Llenguatge C++

Potser molts es qüestionen la raó per la qual utilitzar C++, bé, caldrà a comentar les diferents facetes que té. Un punt molt important és que aporta tots els beneficis del C [11] (Kirch-Prinz, et al., 2002):

- És eficient, molt proper al llenguatge màquina i dóna lloc a programes compilats, és a dir, no s'han d'interpretar en temps d'execució.
- Molt portable, fàcil de canviar entre plataformes depenent de les opcions del compilador.
- Capacitat d'aprofitar un gran nombre de llibreries extenses.

A part del comentat anteriorment, afegeix la possibilitat de la programació orientada a objectes. El problema de la programació tradicional és que el desenvolupador ha d'assegurar-se que les dades estan correctament inicialitzades abans del seu ús i si la representació d'aquestes dades canvia, les funcions han de ser modificades també. Per tant, aquesta característica ens ofereix el següent:

- Reduir la possibilitat d'errors, l'objecte propi controla l'accés a les seves dades.
- Gran capacitat de reutilització, una classe pot ser utilitzada en diversos programes fàcilment.
- Poca necessitat de manteniment, un objecte pot modificar la seva estructura interna sense haver de canviar l'aplicació.

És més, per tenir una idea aproximada del percentatge d'ús a tot el món d'aquest llenguatge, s'ha pres com a referència l'índex més emprat en aquesta tasca, extret de l'informe que comparteix nom amb l'índex, anomenats TIOBE. Aquest valor s'actualitza cada mes, pren com a referència els 25 millors cercadors d'internet, han de tenir un marcador de coincidències, han de presentar els valors en HTML i no ser una web pornogràfica. Els llenguatges que es valoren han de tenir entrada a Viquipèdia, passar el test de Turing i tenir més de 5.000 entrades a Google si s'introdueix "<llenguatge> programming" [12] (TIOBE, 2018).

L'octubre de 2018, data actual del projecte, el C++ és el tercer llenguatge més utilitzat amb un 7,693% de tots els llenguatges valorats, només tenint per sobre el Java i el C com a primer i segon respectivament.

Com a contrapartida, el fet de ser un llenguatge que ha de ser compilat fa que sigui més lent de testejar (cada canvi al codi suposa haver de recompilar el programa). Un altre defecte

que té és el grau de complicació per escriure, no és tant intuïtiu com altres llenguatges com Python però és explícit per ser òptim. En contrast, també permet amb aquesta dificultat fer patrons de disseny més complexes.

Descripció del projecte

3.1 Introducció

Aquest capítol se centra a detallar com utilitzar les eines ja esmentades per evitar problemes i millorar certes facetes del procés. Primerament, se centrarà amb l'explicació d'elements bàsics del VectorCAST. A continuació, es cobrirà les bones pràctiques a l'hora de testejar amb aquest programa. A partir d'això, tractaré la resolució de problemes concrets que a força d'investigar s'han pogut solucionar o no. En tot aquest procés es veuran aspectes a tenir en compte pel C++.

Finalment, s'acabarà parlant de les configuracions del Jenkins. Aquest capítol explica l'experiència de cobrir unes 130 classes en C++ en uns 6 mesos. Per trobar tota la reproducció del projecte amb la que he explicat beneficis, configuracions i problemes del VectorCAST i el Jenkins un es pot dirigir a la secció tfg del meu Github.

3.2 Ús bàsic del VectorCAST

3.2.1 Interfície VectorCAST

Un cop obert el programa, el primer pas a fer és familiaritzar-se amb els seus components visuals principals com el de la Figura 3.1:

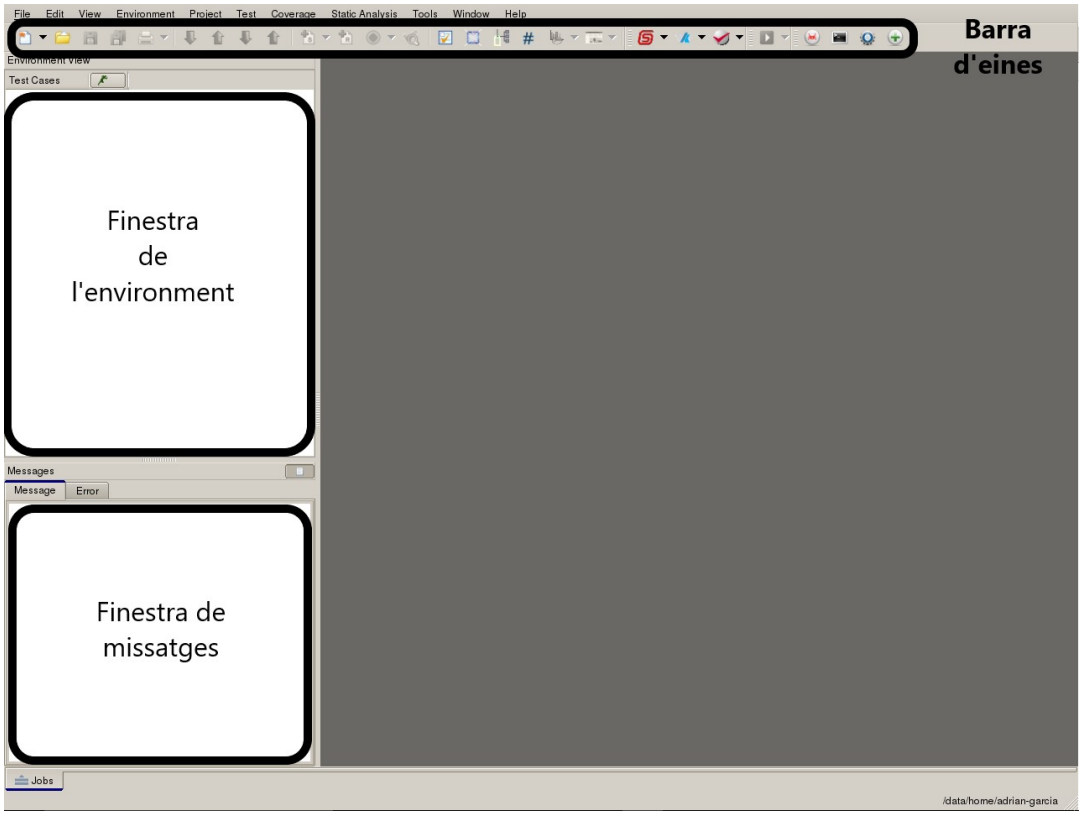


Figura 3.1. Interfície VectorCAST[7]

[7] Imatge pròpia

Barra d'eines
Amb aquesta secció de la finestra podrem accedir als entorns creats, obrir diferents informes, veure variables, entre d'altres. Aquí és on es troba el gran gruix d'opcions amb les quals s'interactua amb el VectorCAST.

Vista de l'environment
Aquí es troba l'estructura de l'entorn de treball, en un de simple destaquen les seccions d'initialization on es creen els objectes base que es poden fer servir als tests de cada mètode de la UUT, els compound test que aconsegueixen agrupar dos o més tests seguits (utilitzats majoritàriament per les màquines d'estat finites) i la secció de les UUT amb els seus mètodes on els afegim els seus respectius tests unitaris.

Finestra de missatges
On hem de veure cada procediment que es crida en tot moment al VectorCast i diu on trobar els errors. Aquestes accions són el resum de totes les ordres de VectorCAST que podem veure a la finestra de treball plegada a sota d'aquesta.

3.2.2 Fitxers bàsics

Tenint en compte que aquest tipus de plataformes utilitzen normalment diferents fitxers i directoris trobo important tenir a mà una descripció bàsica del que es pot trobar a cadascun d'ells pel cas que hàgim de cercar o modificar informació en un moment donat.

Nom_environment.env
Un document amb les instruccions necessàries per construir el nostre environment.

Nom_environment.vce
El fitxer amb el qual obrim el nostre environment a través del VectorCAST.

CCAST_.CFG
On es guarda la configuració que pot ser accedida o modificada des de la secció Tools > Options. En el cas que es vulgui sempre la mateixa configuració podem copiar el fitxer als directoris de treball on utilitzem el VectorCAST per no haver de canviar-ho a mà cada cop que comencem un nou environment.

Directori Nom_environment
Aqui es troben continguts el codi font dels test harness, el harness executable, els fitxers amb l'User Code, i els fitxers amb dades de l'environment. Cap dels mencionats anteriorment haurien de ser directament modificats o oberts per l'usuari en situacions normals d'ús. Si són modificades incorrectament, pot esdevenir un comportament imprevisible o la pèrdua de dades com a resultat.

3.2.3 Creació d'un environment

Un unit test environment és un directori que conté fitxers utilitzats durant el procés de test (bases de dades amb l'estructura de les unitats, fitxers font del test harness i carpetes amb resultats).

El procés de construcció d'un environment està compost de les següents parts:

1. Parseig del codi font de les unitats que són testejades i les seves dependències
2. Determinació de les unitats a stubejar
3. Composició automàtica del driver i els stubs
4. Compilació dels components i enllaç amb les UUT

Per crear un nou environment bàsic hem de seguir els següents passos:

1. Clicar a la barra d'eines a la secció File > New > C++Environment o a la icona d'un full blanc amb una estrella.
2. Escollir un compilador (el de la màquina final on s'executarà el programa).
3. Donar nom a l'environment.
4. Seleccionar Traditional unit testing method.
5. Seleccionar l'ús de whitebox.
6. Proporcionar el path al codi font
7. Escollir les UUT.
8. Si es considera necessari afegir User Code creant classes o variables que s'utilitzaran més endavant.

L'ús de whitebox en C++ ens permet tenir accés als atributs i classes de l'àmbit privat d'una classe. Això ens facilita molt la feina, ja que per posar a punt definint directament un atribut que no veuríem perquè és privat (només pot ser accedit mitjançant la crida de funcions).

3.2.4 Gestió dels Test Case a un mètode

Clicant amb el botó dret sobre un mètode es pot afegir o un cas personalitzable des de 0 o afegir els Basis Path perquè et generi automàticament tots els possibles camins del programa.

Un cop afegits, es poden modificar les classes, entrades i resultats esperats creats. Un cop ja està configurat, es pot procedir a executar el test amb el botó F5 o el dibuix d'un quadre lila amb un triangle blanc al mig (situat a la barra d'eines).

Al finalitzar l'execució, sortirà l'informe on es pot observar les crides que han tingut lloc (expressats en esdeveniments o events en anglès) tant a funcions com stubs, si els resultats esperats coincideixen o no i quins han sigut específicament. Si hi ha algun warning, excepció o signal termination també es mostra aquí mateix.

En el cas que algun paràmetre sigui User Code, no es mostrarà al report, sinó que s'haurà d'escriure des de l'User Code al standard output per fer aparèixer el resultat al report.

3.2.5 Regression scripts

Un cop acabat el test d'una classe, es procedeix a crear els Regression scripts (Environment > Create Regression scripts). Aquests serveixen per no haver de conservar cada directori i fitxer .vce que té cada classe testejada, que són considerablement més pesats.

Quan es creen, es generen tres fitxers:

- Un executable .csh o .exe depenent si s'està utilitzant Linux/Mac o Windows respectivament. Aquest té les ordres per reproduir i executar el test que s'ha fet anteriorment generant tot seguit l'informe amb els resultats.
- Un document .env on es troba la configuració de l'entorn a reproduir.
- Un arxiu .tst on es troben les inicialitzacions d'objectes, entrades i resultats esperats de cada test.

Aquest conjunt ocupa molt menys espai i fa la recerca d'informació de l'entorn molt més fàcil pel qui treballa amb ella. Aquests són els fitxers que es pugen al SVN, a part, s'afegeixen a un projecte VectorCAST Manage per recollir-los tots i mostrar-los al Jenkins.

3.3 Bones pràctiques

3.3.1 Ús de static

El problema que s'introdueix a continuació pot passar amb qualsevol tipus, però en particular, sempre passa amb els punters de C++ o els de la llibreria estàndard. Quan el punter s'utilitza entre diferents funcions, és a dir, que hi ha canvis de scopes, per tal d'optimitzar l'ús de memòria el punter es buida. La solució és afegir l'atribut static en la creació de la variable.

Aquesta situació és especialment molesta amb els casts de C++ d'un tipus a un altre i amb callbacks, que en trobar un punter buit aturava el programa.

3.3.2 Fer ús de l'INIT

Per no haver de configurar a cada cas de test l'objecte que s'utilitza el més aconsellable és crear una classe base (i utilitzar-la a tots els tests). L'INIT es pot crear al visor de l'environment frent click amb el botó dret a la part d'initalization. Cal assegurar-se que fent clic amb el botó secundari l'opció d'automatic initalization estigui marcada. A la següent imatge es mostra l'exemple d'un INIT extret de la reproducció del que es feia al projecte de l'empresa (Figura 3.2).

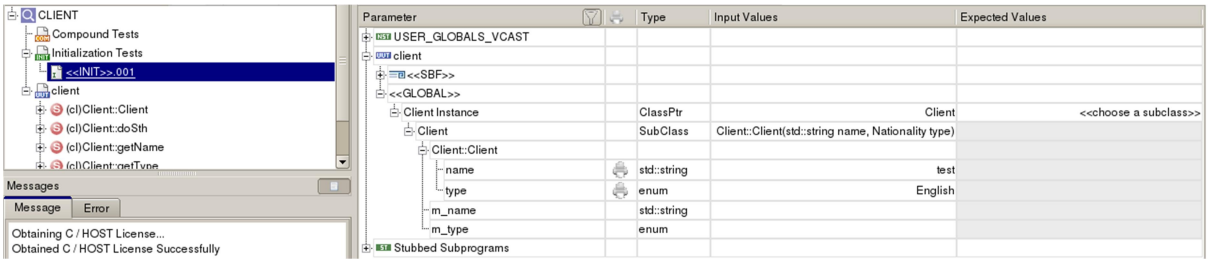


Figura 3.2. Creació d'un INIT[8]

[8] Imatge pròpia

3.3.3 Donar un cop d'ull a les Unstubbed Functions

Hi ha vegades, que per defecte segons quines funcions no es marquen per stubbejar. Per saber si és possible fer ús d'elles, podem anar a: Environment > View > Unstubbed Functions. Si figura allà, podem procedir a actualitzar l'entorn i afegir la funció a Choose Unit Under Test dins la pestanya Additional Stubs o Library Stubs depenent de si és una llibreria o no. Un exemple més visual és mostrat a la Figura 3.3.

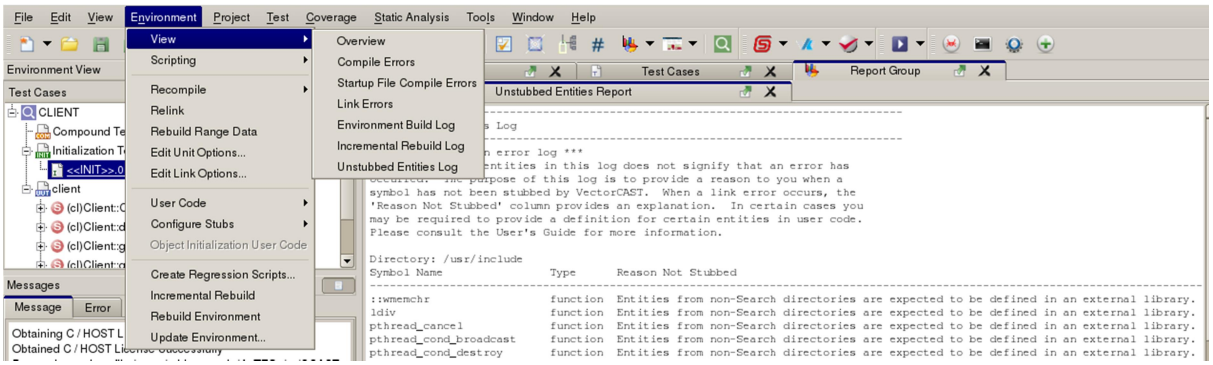


Figura 3.3. Observació de les Unstubbed Functions[9]

3.3.4 Redirigir el standard output

Per tal de veure variables que són de l'User Code o comprovar si es crida el stub propi o el del Vectorcast s'ha d'activar a l'apartat Options > Report > Redirect standard output. Així dins de l'informe d'execució ens apareixerà aquesta informació. A la Figura 3.4 ja s'han deixat marcades com exemple.

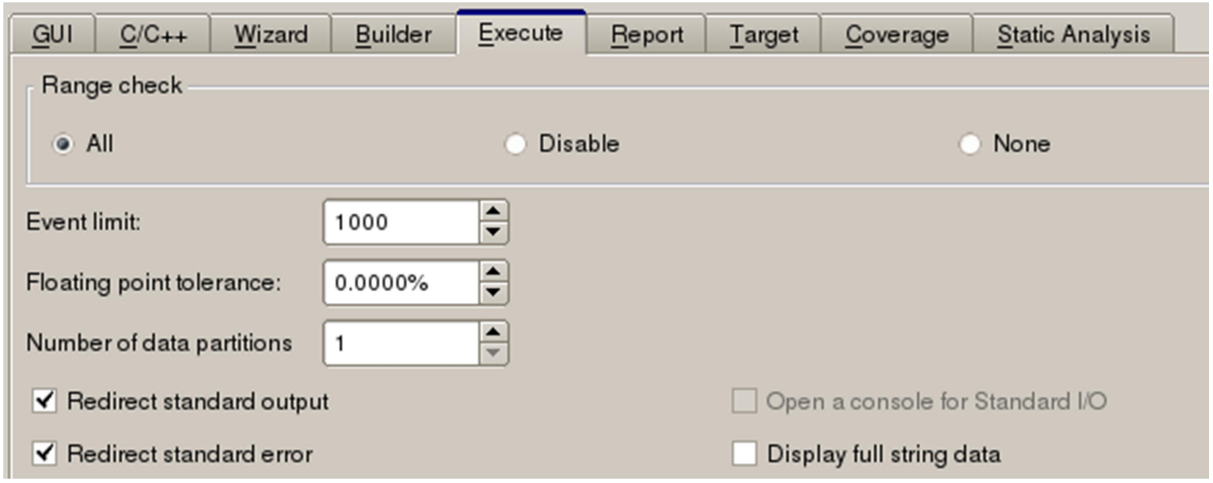


Figura 3.4. Desviació del Standard Output[10]

3.3.5 Fer ús del Debugger

Si el que es vol és informació concreta de l'execució d'un mètode, una manera ràpida i fàcil de fer-ho és amb el Debugger. Per utilitzar-lo, només s'ha de fer clic al botó dret del ratolí sobre el cas del mètode que es testeja i seleccionar Execute with debugger. Per si no ha quedat clar, la Figura 3.5 mostra visualment el cas.

[9] Imatge pròpia
[10] Imatge pròpia

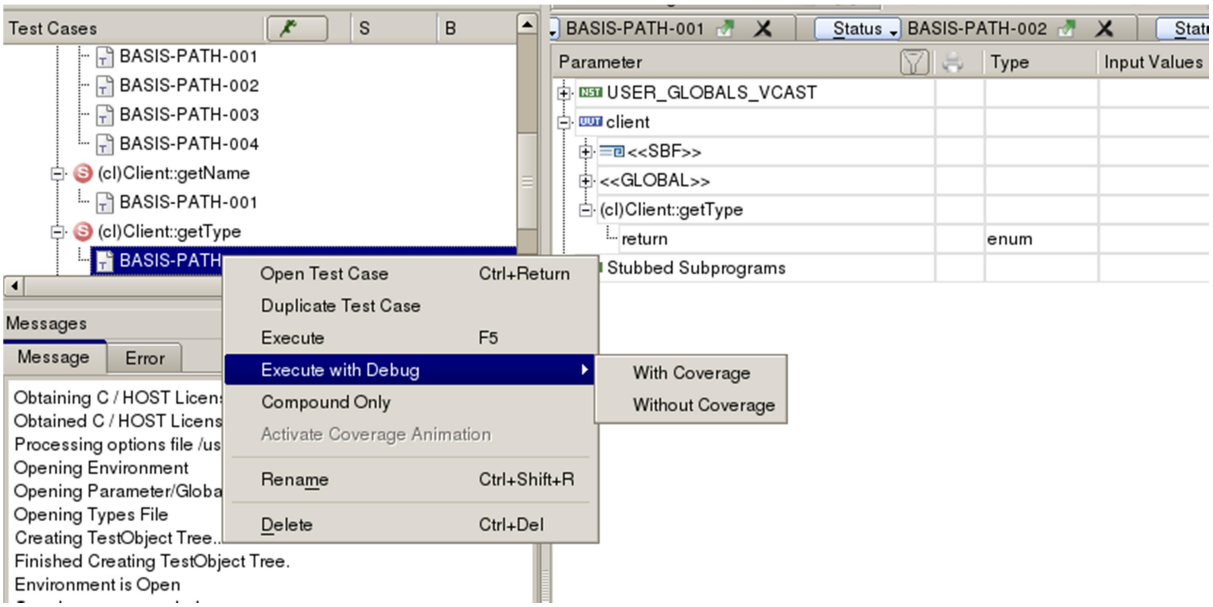


Figura 3.5. Com accedir al Debugger[11]

Amb aquest terminal (amb l'aspecte de la Figura 3.6) es pot veure línia per línia l'execució de la funció, es pot entrar a crides dins d'aquesta i mostrar variables gràcies als breakpoints (si no s'aconsegueix posar pel nom del mètode, també es pot posar per número de línia). La manera d'introduir breakpoints és la següent:

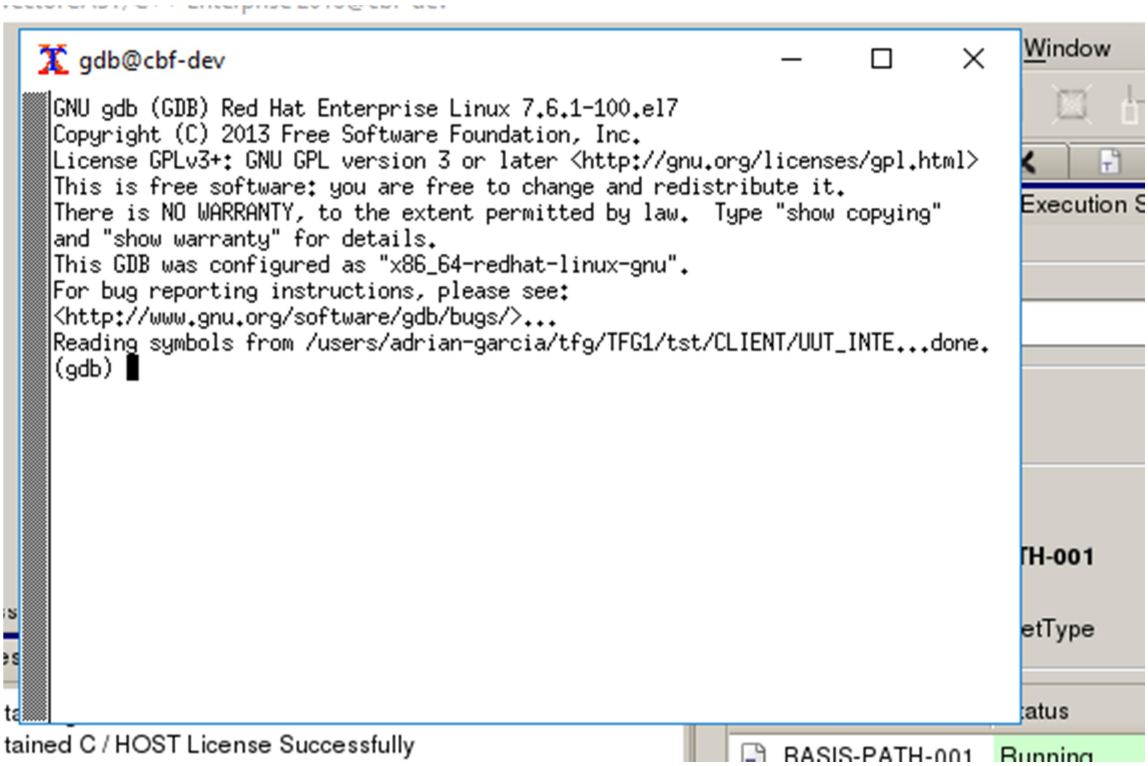


Figura 3.6. Terminal Debugger VectorCAST[12]

[11] Imatge pròpia
[12] Imatge pròpia

```
b Namespace::Class::function(parameters)
b 11480 // breakpoint a la línia 11480
p var //print var
```

També és molt útil activar el control d'excepcions per veure exactament on està l'error i de què es tracta amb la comanda catch throw. Un cop activats els breakpoints o el catch throw mitjançant l'ordre run i next (també r i n) s'executa el test del cas i s'avança entre les línies de codi.

3.3.6 Localitzar threads en els tests

Sempre que es treballi amb threads s'ha d'intentar stubejar la part del codi que crea un nou thread. La raó és senzilla, si no es fa com s'aconsella, el programa principal pot acabar correctament i marcar tot el codi com cobert i el thread desmarcar-lo fins on arribi.

Per localitzar threads, que no sempre es veu mirant el codi a simple vista, es pot veure en el cas que es quedi esperant que acabi el thread obert. Si l'execució d'un cas mai acaba i a la part inferior esquerra de la interfície apareix el missatge UUT_INST en groc.

3.3.7 Consultar els fitxers .tst anteriors

Molt sovint és normal que diverses classes comparteixin paràmetres d'entrada o atributs que s'han d'entrar com a User Code. Aquí hi ha dos problemes: que de vegades no és possible recordar com s'ha declarat cada peça de codi que s'introdueix al programa i que hi ha parts d'User Code bastant extenses.

En comptes d'escriure cada cop la inicialització d'aquests components, llegir l'arxiu .tst és de gran ajuda. Allà es poden veure totes les entrades i inicialitzacions de classes que s'han dut a terme en un entorn. Un cop localitzat el segment de codi que es necessita és tan fàcil com copiar-lo a l'entorn en el qual es treballa. En estar tan localitzat el codi que es busca resulta molt còmode per reutilitzar codi de casos passats.

3.4 Resolució de problemes

3.4.1 Desaparició de l'informe després d'una execució

En certs moments, quan s'afegeix User Code o es modifica molt l'entorn quan s'executa un cas o diversos apareix un error d'execució i no mostra cap informe de què succeeix. Això es deu a un error dins del codi afegit per l'usuari, que fa que no compili i no es generi cap resultat. A la Figura 3.7 queda clar que no es pot accedir al report.

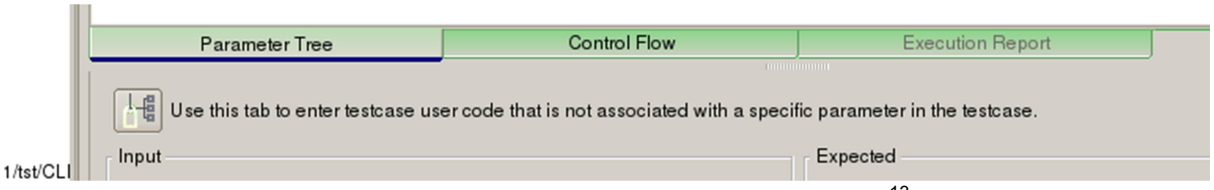


Figura 3.7. Desaparició del Execution Report[13]

[13] Imatge pròpia

3.4.2 Retorn de diferents valors als stubs

Hi ha mètodes que criden la mateixa funció diverses vegades dins la seva execució. El més lògic és que es vulgui que retorni valors diferents cada cop. Si és stubejada, es pot variar el valor escrit a la cel·la separat per comes per tipus simples (int, char, bool...). En el cas que sigui via User Code, s'ha de fer amb un if que avaluï i modifiqui una variable del VectorCAST:

```
static shared_ptr<X> p1 = std::make_shared<X>("test");
static shared_ptr<Y> p2 = std::make_shared<Y>("test");

if (VECTORCAST_INT_1 == 0) {
    <<count.function.return>> = &p1;
    VECTORCAST_INT_1++;
} else {
    <<count.function.return>> = &p2;
}
```

3.4.3 Detecció d'una classe dins un namespace

Una de les característiques del C++ és poder diferenciar dues classes, variables o funcions amb el mateix nom pel seu namespace. El cas és que escrivint User Code pot aparèixer un error de compilació dient que la classe, variable o funció no pertany al namespace que s'ha escrit.

Aquesta és la conseqüència de no trobar el prototip del header on es descriu que pertany a aquest namespace. No té cap més solució que afegir l'include del .hpp de la classe on es declara aquesta informació dins de l'Appendix User Code.

3.4.4 Redirecció del standard input

Si el programa que s'està testejant conté una lectura del standard input per defecte (el teclat) no es podrà fixar el paràmetre d'entrada correctament.

Primer es va provar la possibilitat de fer servir correctament la solució proporcionada a les opcions de VectorCAST (consistent en redirigir l'entrada a un fitxer). Veient l'impossibilitat de fer que funcioni correctament, es va procedir a redirigir l'entrada per User Code. Al cap de diversos intents es va aconseguir amb la següent línia de codi:

```
freopen("input.txt", r, stdinput);
```

Aquesta ordre fa que es llegeixi el contingut del fitxer input.txt com a estímul de la funció que s'està cobrint. IMPORTANT: no és l'única manera en C++ de redirigir el canal d'entrada, però és l'únic efectiu per VectorCAST.

3.4.5 Problema de creació d'un objecte concret a partir d'un abstracte

Quan es planteja la necessitat que es planteja al títol d'aquest apartat afegint la nova classe a l'Appendix Code de l'User Code de l'entorn hauria de ser suficient. El problema és que per defecte hi ha una opció del VectorCAST que automàticament genera la classe concreta i entra en conflicte amb la feta per l'analista de tests.

Aquest error es pot detectar fàcilment quan es veu al compilador el missatge de Duplicated call to function … . La solució és desmarcar la casella “Autogenerate concrete classes from abstract classes”, situada a Opcions > Builder (Figura 3.8).

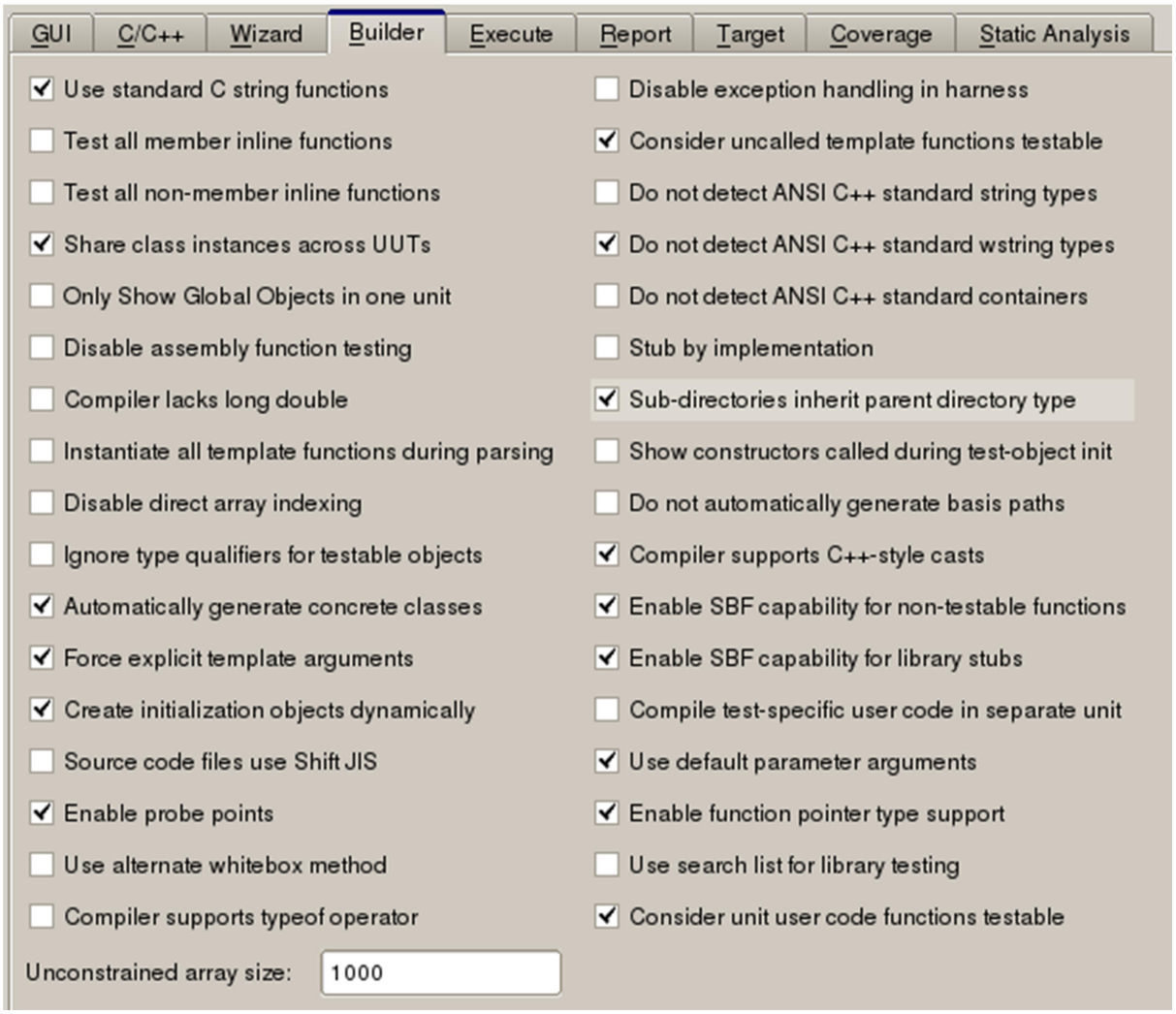


Figura 3.8. Menú d’opcions[14]

3.4.6 Problemes amb els shared_ptr

Tractant amb atributs que són del tipus shared_ptr de vegades sorgeix un segmentation fault al debugger relacionat amb la llibreria shared_ptr.

La raó per la qual el programa falla és que no detecta que la classe sigui l’owner (posseïdor) del shared_ptr. La solució és anar a la inicialització de l’objecte i en comptes de fer servir la interfície, crear un shared_ptr apuntant a l’objecte i passar-li a l’entrada:

```
static std::shared_ptr<Class> p =
std::make_shared<CBF::Class>(parameter);
<<X instance>> = p.get();
```

[14] Imatge pròpia

3.4.7 Impossibilitat de stubeig de funcions Template

El C++ ofereix la possibilitat d'utilitzar Templates tant per funcions com per objectes. Aquesta peculiaritat permet a partir de la definició d'un sol cos (classe o mètode) adaptar-se a més d'un tipus de paràmetres sense haver de repetir el codi sencer.

Malauradament, segons el suport tècnic de VectorCAST aquesta funcionalitat encara no és compatible amb la configuració de stubs, és a dir, no es pot evitar que es cridi la funció original substituint-la pel comportament volgut.

3.4.8 Stubeig de classes Template

Aquesta tasca pot semblar en primera instància trivial, ja que VectorCAST facilita les instruccions a la seva documentació. Tanmateix, seguint els passos que s'explicaran a continuació es va trobar una situació que va suposar una pèrdua considerable de temps.

No val només a posar-la com a UUT, sinó que també cal afegir la següent línia amb User Code:

```
template class X<type>;
```

On X és l'objecte i type qualsevol tipus que accepti la template. Després de configurar adientment l'User Code l'entorn seguia sense mostrar cap UUT que testejar. Era molt difícil saber per on tirar, ja que no hi havia cap missatge d'error ni res per l'estil. Després de consultar amb el suport tècnic més de dues setmanes es va determinar que faltava marcar una casella d'opcions (anomenada test all inline functions).

3.4.9 UUT amb dependències a altres classes

Aquest cas és amb diferència el que més guerra ha donat. Quan hi ha un objecte en què intervenen diverses classes el VectorCAST no troba les declaracions dels mètodes situats als .cpp de cadascuna. Això provocava la impossibilitat de la creació de l'entorn donant errors al linker.

Consultant amb el suport de VectorCAST es va proposar d'afegir a l'User Code > Appendix Code els includes dels .cpp que no trobava el compilador. Aquesta primera solució tenia dos grans problemes:

- Els temps de creació/actualització de l'entorn i els temps de compilació de l'User Code s'allargaven a períodes insostenibles.
- Impossibilitat de stubejar les classes incloses.

La primera conseqüència es deu a l'extensió del .cpp de la UUT. Tot el que es posa a l'Appendix Code s'afegeix just a sota del codi font de la UUT. Llavors, si se li afegeixen tres includes, s'està triplicant aproximadament el codi que estem testejant i així respectivament. Recordar que per canviar o crear un entorn s'ha d'analitzar tot el codi font, veure les dependències, determinar què es pot stubejar i instrumentar el codi. Tot l'esmentat anteriorment es veu afectat.

Respecte a l'efecte sobre els stubs cal dir que tot el que es trobi afegit al codi font no ho intentarà stubejar si no és propi de la classe analitzada.

Per tal de millorar l'espera, es va recórrer a incloure com a UUT els objectes dels quals depenia la classe testejada. D'aquesta manera, sense tenir un gran impacte quan es canviava o creava l'entorn, sí que va millorar molt el temps de compilació de l'User Code i es va poder stubejar les classes afegides.

Malgrat això, quan es miraven els resultats del test al Jenkins encara que estigués coberta al 100% el percentatge mostrat disminuïa. En aquest cas, la ràtio es feia tenint en compte les UUT afegides de més, que no havien estat cobertes. La solució va ser senzilla, anant a l'apartat Coverage > Initialize Custom (Figura 3.9) es podia desmarcar quines sortien a l'informe i així veure el percentatge d'una, a la Figura 3.10, si es vol testejar client, es desmarcarien les classes manager i manager_driver.

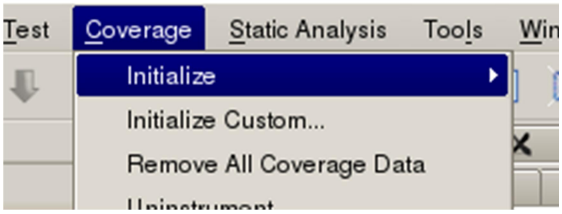


Figura 3.9. Situació Initialize Custom[15]

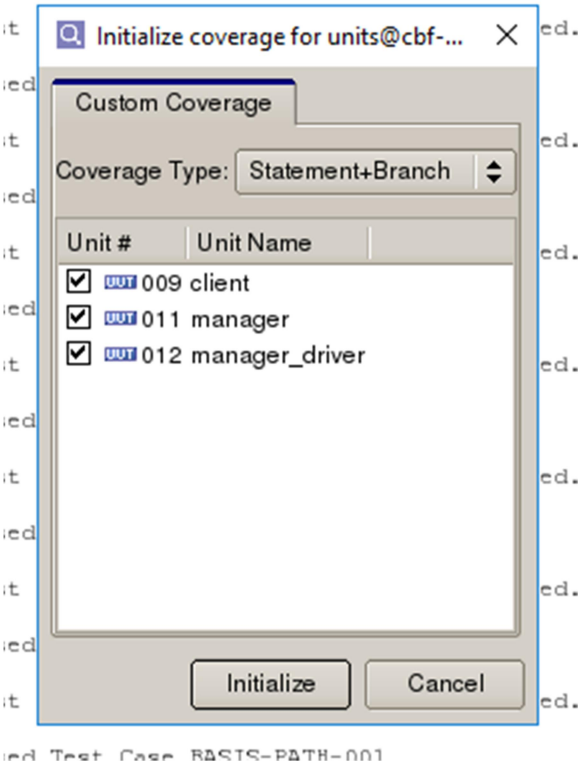


Figura 3.10. Caselles a desmarcar[16]

Abans de donar per bona aquesta solució, es va buscar millorar el temps de reproducció/creació/actualització de l'entorn. Si al quadre on s'escullen les UUT, a Stub dependències en comptes de l'opció all s'escull custom, es pot triar quines UUT es poden stubejar i quines no. Fent aquesta distinció es va aconseguir un canvi realment considerable, es veurà concretament a la discussió de resultats.

[15] Imatge pròpia
[16] Imatge pròpia

3.4 Configuració del Jenkins

Per tal de no mostrar les configuracions del Manage, les del Jenkins i el codi de l'empresa s'ha reproduït un seguit d'entorns VectorCAST, un projecte Manage que els englobi i els pugui passar a un Job de Jenkins.

En aquest punt es mostraran els passos bàsics per crear tot l'esmentat al paràgraf anterior i els obstacles que s'han trobat en aquest procés.

3.4.1 Creació d'un projecte Manage

A VectorCAST, al principi de la barra d'eines, al costat de la icona per crear un entorn hi ha un desplegable que et deixa triar crear un entorn o un projecte Manage.

Un cop seleccionat, només cal posar nom al projecte i afegir els entorns creats prèviament. Els entorns poden ser Regression scripts o entorns actius (.vce). A part d'afegir-los es poden dividir en grups si es vol fer alguna diferenciació, entre dos projectes per exemple.

Ara ja es poden executar i editar els entorns per més tard veure els informes resultants. Aquest és l'última peça que cal per executar el VectorCAST al Jenkins i serà qui li doni els reports que ensenyarà al navegador.

És molt important donar-li permisos d'escriptura i execució als fitxers resultants, en cas contrari, es tindran problemes quan el Jenkins intenti interactuar amb els fitxers.

3.4.2 Creació d'un VectorCAST Job a Jenkins

Al menú principal es pot observar a la columna esquerra el nom de VectorCAST i el seu símbol.

Entrant dins de l'enllaç se selecciona l'opció Create Single Job.

Aquí és on es configura la majoria de paràmetres del Job:

- S'ha d'especificar el path on es troba el projecte Manage (.vcm).
- Reintent d'identificació de llicència en cas d'error.
- Fer ús d'un script per Windows o Unix, dins d'aquest script s'han d'exportar dues variables del sistema, el path on es troba instal·lat el VectorCAST i la IP del servidor de llicències.
- Entre les diferents opcions es deixaran per defecte, es vol que generi tots els reports i que siguin en format html per poder-los visualitzar al Jenkins. El nivell d'error també es vol activat (Unstable) per tal de veure si ha fallat o no.
- Es pot afegir un control de versions si es desitja. En el cas d'aquest exemple, com que s'havia de crear un repositori dins la xarxa local de l'empresa no s'ha fet per falta de permisos. En canvi, pel projecte real es descarreguen les actualitzacions des del SVN cada cop que s'executa el Job.

S'acciona el botó de Create per acabar la configuració preliminar. Ara es procedeix a entrar dins del Job i modificar la configuració general d'aquest.

Es pot afegir una descripció del Job, restringir l'accés a la documentació resultant com a la Figura 3.11, preservar només un nombre finit d'execucions de manera semblant a la Figura 3.12, controlar l'execució amb paràmetres extres, entre d'altres.

Figura 3.11. Elecció d'etiqueta, el Job es desenvolupa al node master[17]

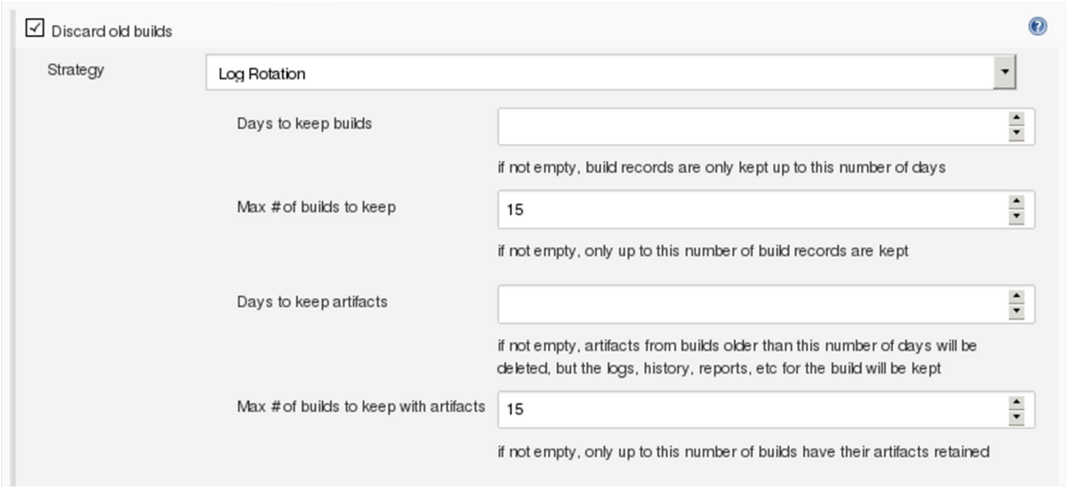


Figura 3.12. Restricció de 15 execucions per guardar[18]

En aquest cas s'ha especificat amb l'etiqueta "master" a quin node es vol executar el Job i que es conservin només les 15 builds més recents.

Si es vol incloure control de versions, és tan fàcil com incloure l'URL amb el path inclòs d'on es troba el directori de treball i les credencials per accedir al Subversion a les franges que es mostren a la il·lustració de la Figura 3.13.

[17] Imatge pròpia
[18] Imatge pròpia

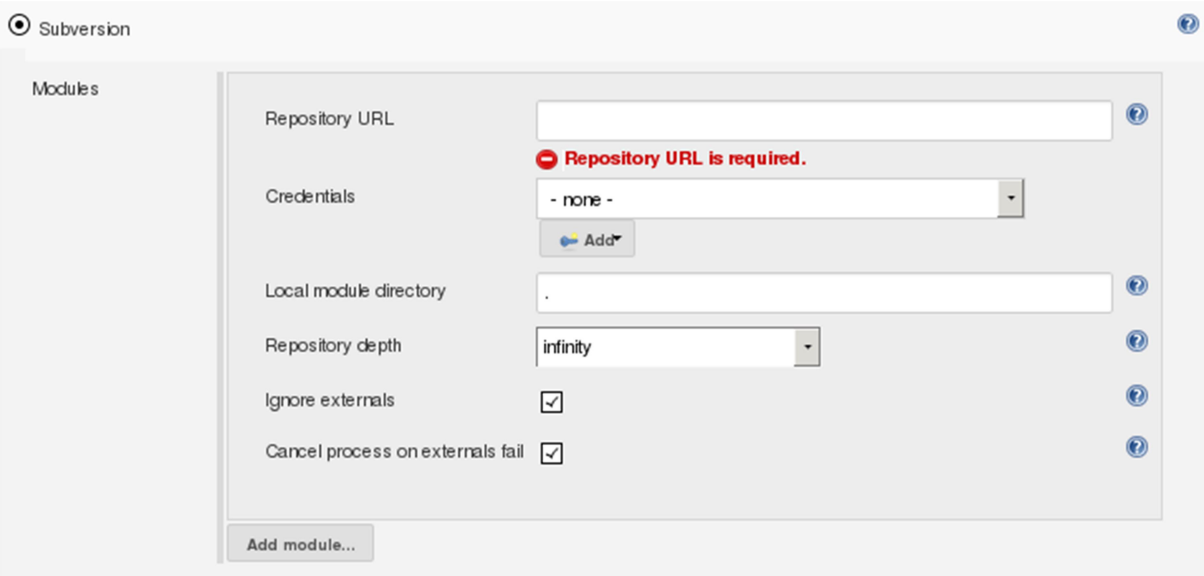


Figura 3.13. Introducció de dades pel SVN[19]

Un dels punts forts dels sistemes d'integració contínua és poder automatitzar l'execució, les que són més interessants que apareixen a la Figura 3.14 són les de Poll SCM i Build Periodically. Amb la primera, si es fa cap actualització al repositori, s'activa el grup de tests. Amb la segona, es pot programar que cada cert temps s'executi sol, aquesta és la que s'ha utilitzat per a les classes reals pel temps que tardava a acabar el job (es feia cada nit per tenir resultats diaris sense cap molèstia).

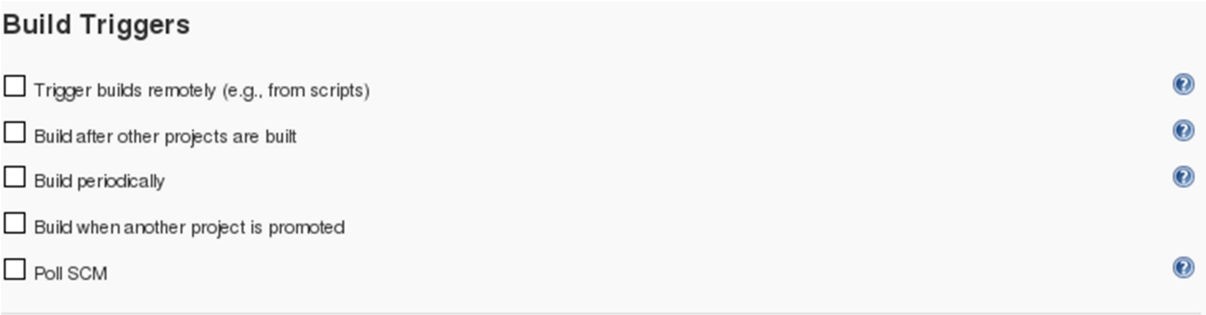


Figura 3.14. Opcions d'activadors automàtics del Jenkins[20]

El resultat de la configuració del Single Job a l'apartat on s'escollia fer l'script per Windows o Unix es troba a la Figura 3.15, aquí es pot acabar de personalitzar cada paràmetre de l'execució del VectorCAST. S'han d'afegir els paràmetres VECTORCAST_DIR amb el path on es troba instal·lat el VectorCAST i el paràmetre LM_LICENSE_FILE que és la IP i el port on es troba el servidor de llicències, el format és el següent: PORT@IP. Important donar un cop d'ull als problemes trobats durant la configuració del Jenkins si s'està intentant reproduir aquest cas.

[19] Imatge pròpia
[20] Imatge pròpia

Figura 3.15. Configuració a partir del Job VectorCAST anterior[21]

Al Post-Build Options es poden afegir els llindars per considerar que una execució ha tingut èxit o no d'una manera intuïtiva com s'aprecia a les Figures 3.16 i 3.17. Amb un dels símbols es té en compte el nombre de tests que s'han pogut executar correctament i el segon el nivell de cobertura respectivament.

[21] Imatge pròpia

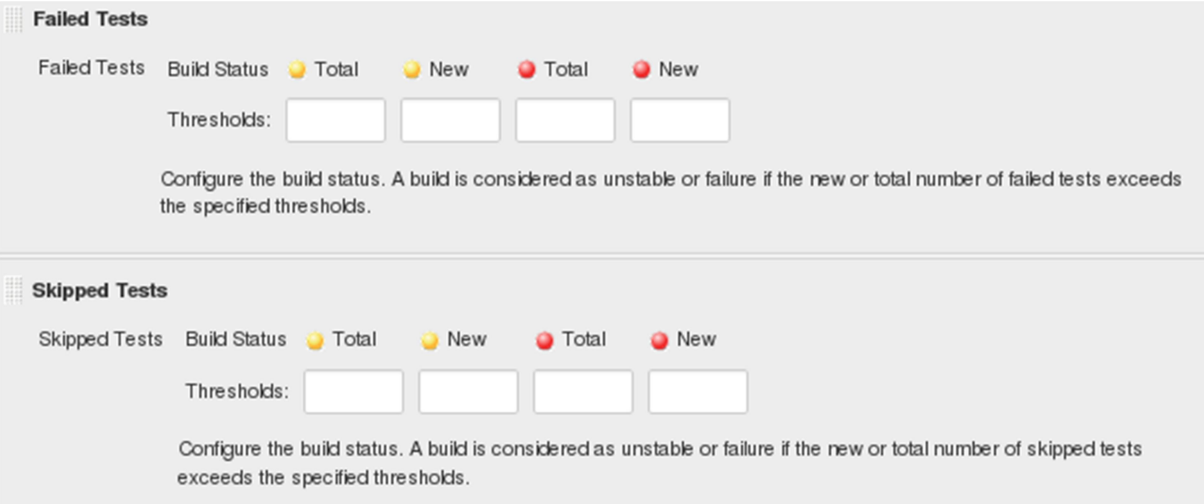


Figura 3.16. Icona de mesura del nivell d'execució[22]

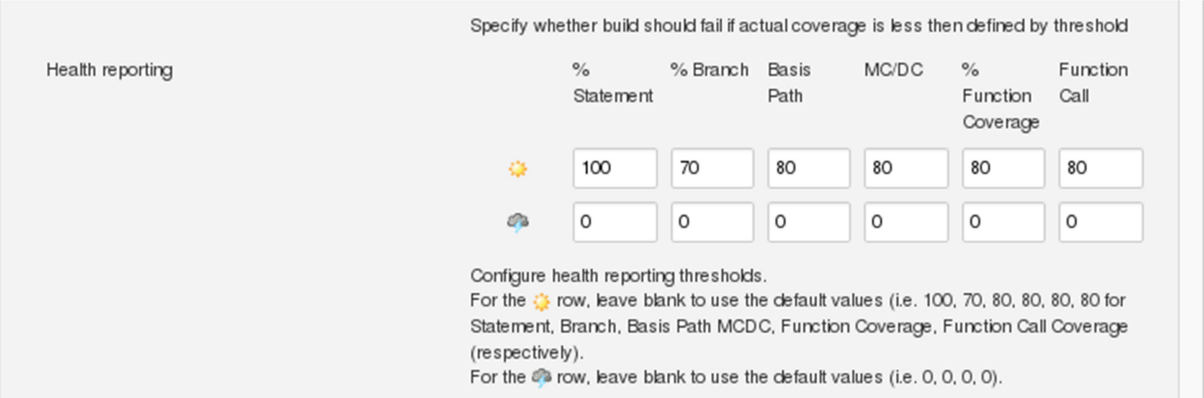


Figura 3.17. Icona de mesura d'aspectes concrets dels tests[23]

3.4.3 Visualització dels resultats

Es mostra un breu resum com a pantalla principal dels Jobs, si no es vol entrar dins d'ells, es pot clicar als símbols per veure els percentatges clau que s'han configurat en el punt anterior.

Accedint a un Job concret es mostra molta informació:

- Es té una gràfica del nombre de tests que s'executen correctament i els que no.
- Un historial de cada execució (build) amb l'enllaç a cada descripció on es té en detall els arxius generats, missatges de la consola durant l'execució, missatges d'error, gràfiques i taules on es mostren els valors de tot el projecte Manage (nombre de línies i condicions cobertes, percentatges de resultats esperats que són correctes i temps d'execució) desglossats per cada objecte.
- Un menú per configurar el Job, mirar els canvis fets al codi font i accedir a més gràfiques.
- Via text hi ha la sortida del terminal d'execució dels tests per debuggar errors.

[22] Imatge pròpia
[23] Imatge pròpia

Tots els informes i gràfiques poden ser descarregats per presentar-los al client. Per no afegir tantes fotos seguides es recolliran els apartats esmentats dins de l'Annex X.

3.4.4 Problemes trobats

Bug en l'script creat automàticament

Si s'inicia un test del Manage tal com s'ha mostrat fins ara, el Jenkins avortarà mostrant un missatge dient que no troba cap fitxer amb el patró de cerca introduït. Aquest error es deu al fet que dins de l'script que s'ha generat automàticament hi ha una línia que li falta tancar l'expressió de cerca amb unes dobles cometes (”). Aquest error només s'ha pogut comprovar si existia a l'script de Unix.

Fitxers fantasma mal esborrats

Amb la creació del projecte Manage, en l'acció de treure els antics environments per posar-ne uns millorats no es van esborrar bé. Aparentment, no dóna cap error visible fins que no es comencen a apreciar valors estranys al Jenkins com els de la Figura 3.18.

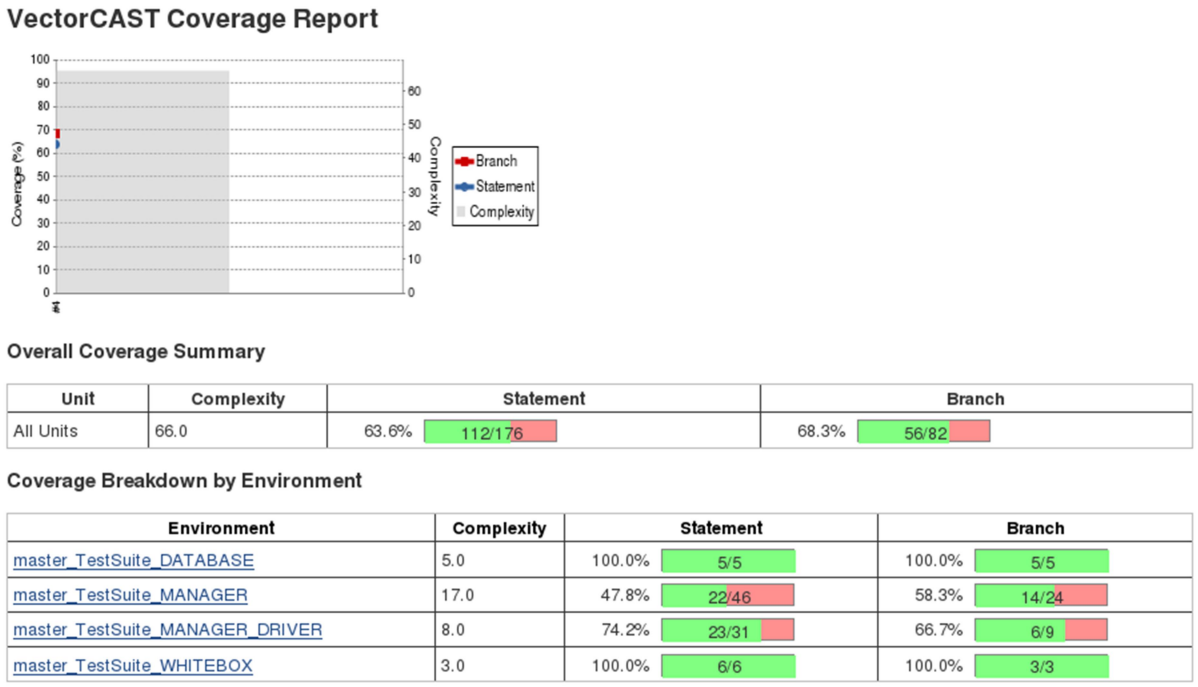


Figura 3.18. Valors sorprenents a la reproducció de l'entorn de tests[24]

Com es pot veure, tot i obtenir tant al projecte Manage com a cada entorn per separat una cobertura del 100%, el Jenkins mostra percentatges diferents i una complexitat (el nombre de statements) el doble del normal. Més tard, investigant al projecte Manage es va veure que hi havia una estructura incorrecta que apareix a la Figura 3.19.

[24] Imatge pròpia

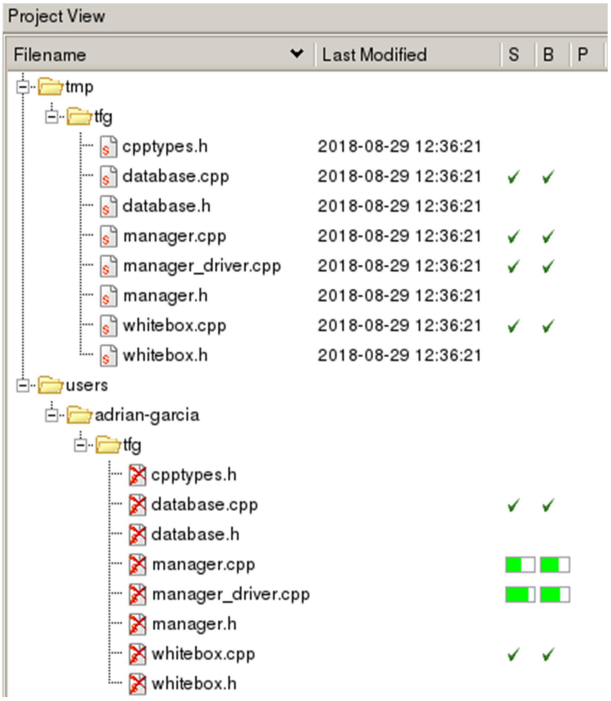


Figura 3.19. Estructura anòmala del projecte Manage[25]

Un cop realitzats els canvis pertinents esborrant aquesta carpeta a la següent execució del Jenkins es pot observar que la complexitat ha baixat (en aquesta gràfica el nombre de statements totals) i la cobertura es marca correctament com es pot veure a la següent imatge de la Figura 3.20.

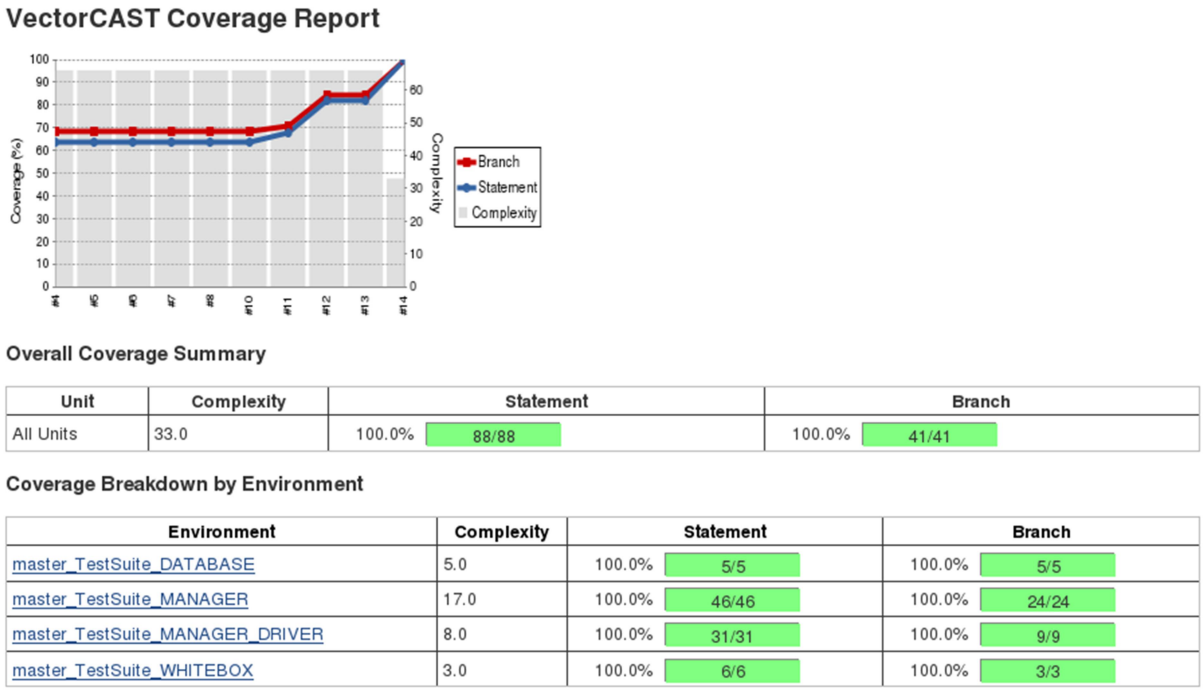


Figura 3.20. Tests després d'arreglar l'error d'estructura de fitxers[26]

[25] Imatge pròpia
[26] Imatge pròpia

Desplegament del projecte

L'objectiu d'aquesta configuració és la protecció de la privacitat, com es mostra a la Figura X s'aïlla el codi a través de dues xarxes amb diferents nivells d'identificació i amb altres mesures de seguretat. A la figura es mostra un esquema aproximat, sense detallar IPs ni mètodes de connexió entre dispositius. Es prosseguirà amb l'explicació de l'esquema de la Figura 4.1.

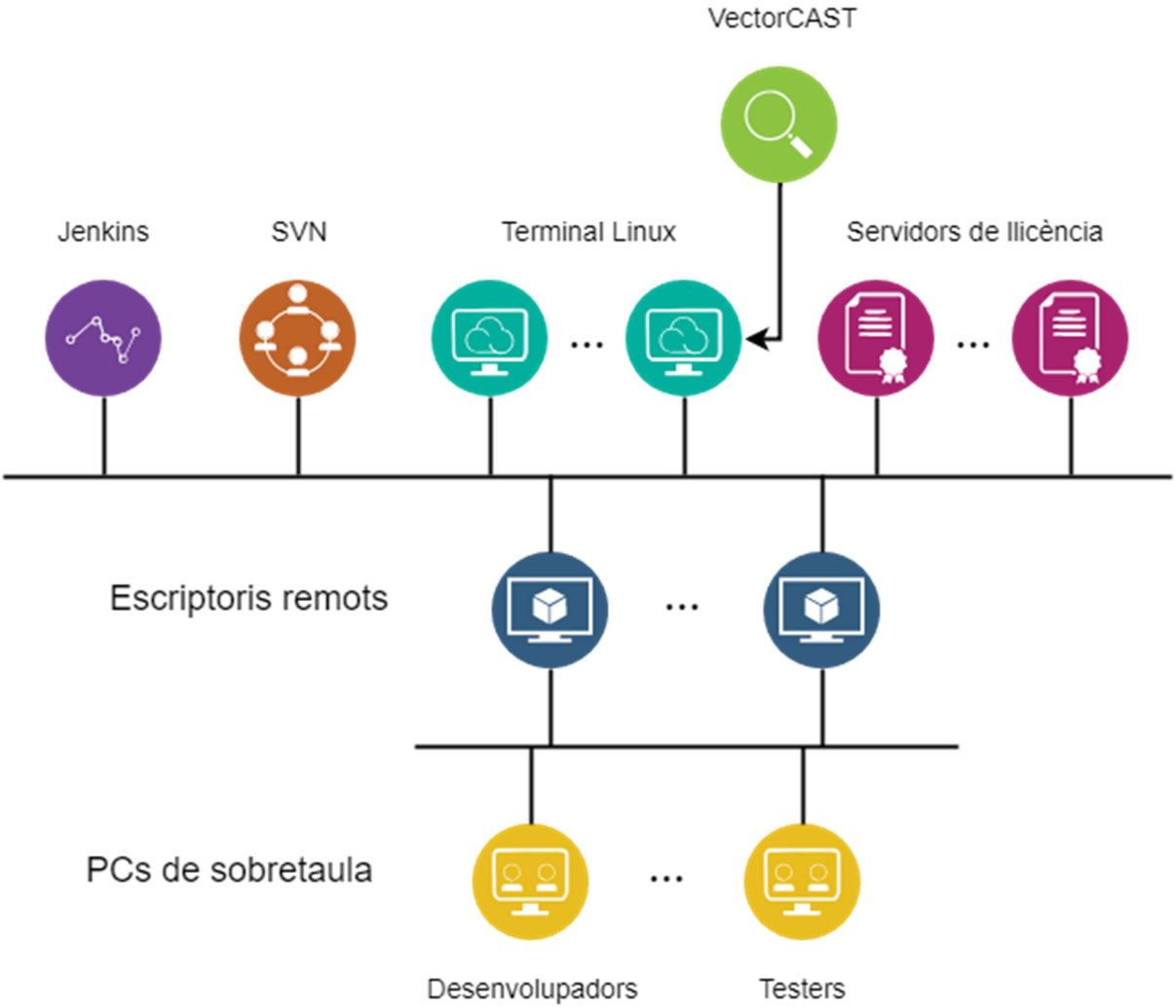


Figura 4.1. Esquema de l'estructura de comunicacions del projecte[27]

Des del PC es connecta amb el primer nivell a un escriptori remot. Aquests via ssh a un terminal Linux amb el qual fent el forward de les X's ens mostra la interfície de VectorCAST a l'escriptori remot. Dins de la seva xarxa té els servidors de llicències VectorCAST, el Jenkins i el SVN amb els que interactua.

Aquesta estructura té com a contrapartida la possibilitat de no arribar al codi des de l'ordinador propi si les comunicacions es tallen en algun moment. Per això, hi ha un equip

[27] Imatge pròpia

encarregat del manteniment del sistema que també gestiona els permisos i sessions que s'utilitzen per a tota l'estructura.

Resultats de l'experiència al projecte

Per tal de valorar el rendiment que s'ha extret a tota la prova s'ha creat una gràfica del nivell de cobertura i s'ha extret una gràfica del Jenkins amb el temps d'execució que s'utilitza per executar tots els tests. La idea és extreure punts significatius que donen suport al motiu del treball i els punts valorats.

Nivell de cobertura dels mòduls principals del projecte CBF
Una primera fase dels tests era arribar a un nivell de cobertura acceptable, un 75% a totes les classes dels dos mòduls. Ja un cop arribat a aquest nivell mínim es prosseguia amb la meta final esmentada anteriorment. Com es veurà el fragment de projecte més representatiu és el TTE_COMMON amb la Figura 5.1. Això és pel fet que és el directori amb més nombre de classes a cobrir i amb el que es va començar a testejar amb el programa VectorCAST.

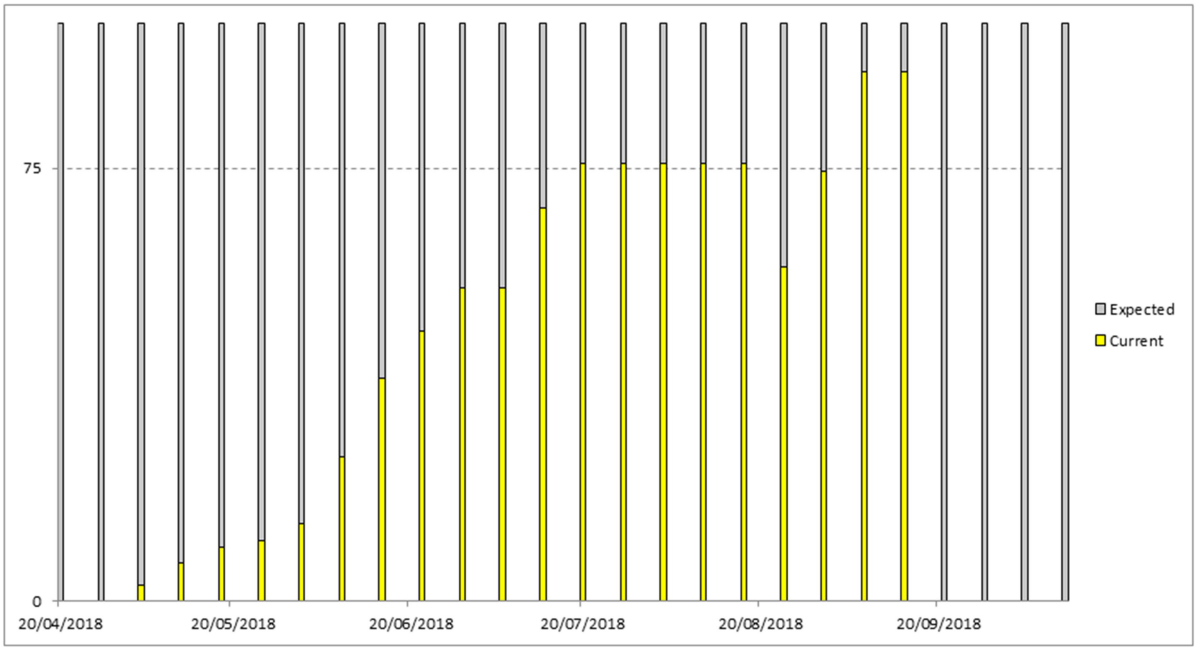


Figura 5.1. Cobertura del mòdul TTE_COMMON durant 5 mesos[28]

Només observant la gràfica a simple vista es poden treure diverses conclusions molt clares.

La primera és que la corba d'aprenentatge tan abrupte presentada pel seguit de problemes concrets fora de la documentació. Per tal d'arribar al mínim de 75% de cobertura va suposar la despesa de 3 mesos dedicats a un únic mòdul.

La segona, es veu si s'observa els valors sostinguts en un període de 2 mesos (entre juliol i agost) en què es pot veure palès la impossibilitat d'avançar en un temps prolongat si no se solucionaven els casos que figuren en aquest treball.

[28] Imatge extreta de GTD

I l'última, que potser és més clara a les estadístiques del TTE_GATEWAY de la Figura 5.2, és que amb la descoberta d'una nova solució a l'hora de passar tests significava un canvi pronunciat en la ràtio de cobertura (d'un 5%, que tenint en compte el nombre de classes que hi ha és considerable). Aquest fet encara reforça més la idea plantejada a la segona observació de la importància de viure aquests casos i saber-los evitar o resoldre.

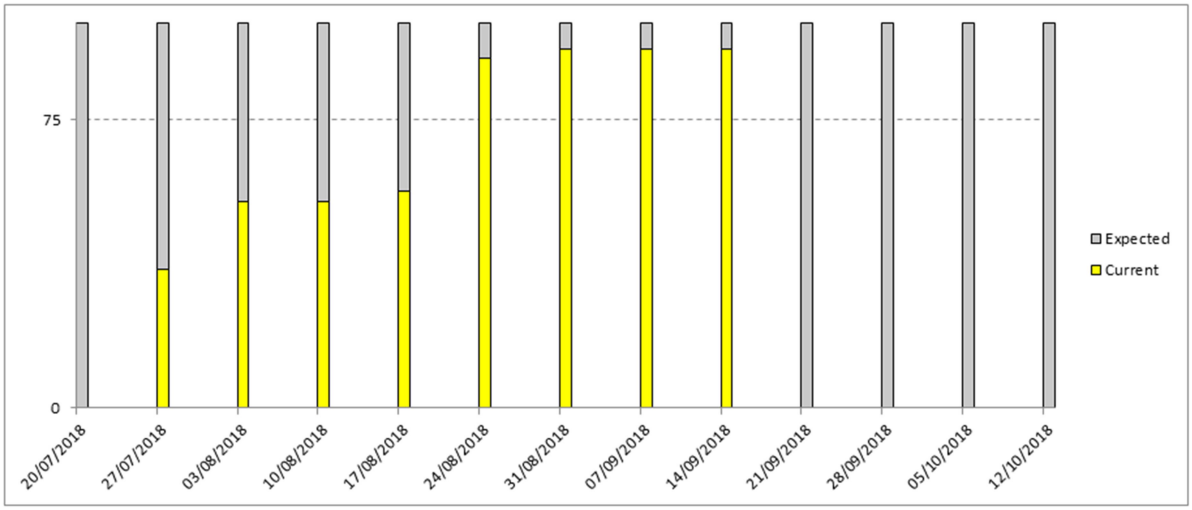


Figura 5.2. Cobertura del mòdul TTE_GATEWAY[29]

El 4-5% que queda per arribar al 100% de tota l'execució del codi ja depèn dels errors trobats al codi que queda en mans dels desenvolupadors. En el cas de trobar-ne un (paràmetres no utilitzats, condicions valorades dos cops o comportaments anòmals), es notificava als desenvolupadors per un portal web de creació de tasques propi de GTD.

Temps d'execució d'un Build al TTE_COMMON

En aquest punt es té en compte el mòdul amb més concentració de classes que cobrir, el TTE_COMMON, que com es mostrarà a continuació ha patit una millora clau pel rendiment de l'equip.

Com es pot veure a la Figura 5.3 els canvis fets amb la personalització dels stubs el punt 3.4.9 no només ha suposat un guany en el nombre de funcions que es poden stubejar sinó que s'ha disminuït el temps d'execució en 3 hores (de 7 hores i 26 minuts a 4 hores i 27 minuts). La columna esquerra mostra el temps concret mostrat al gràfic junt amb un link connectant amb cada execució i el seu detall, la part dreta de la Figura 5.3 ho mostra tot de manera més visual.

[29] Imatge extreta de GTD

Build	↑ Duration	Agent
#81	4 hr 27 min	master
#80	4 hr 39 min	master
#79	4 hr 34 min	master
#78	4 hr 31 min	master
#77	5 hr 14 min	master
#76	28 sec	master
#75	5 hr 9 min	master
#74	5 hr 49 min	master
#73	6 hr 31 min	master
#72	5 hr 49 min	master
#71	7 hr 26 min	master
#70	5 hr 59 min	master
#69	4 min 35 sec	master
#68	3 min 10 sec	master
#67	3 min 12 sec	master

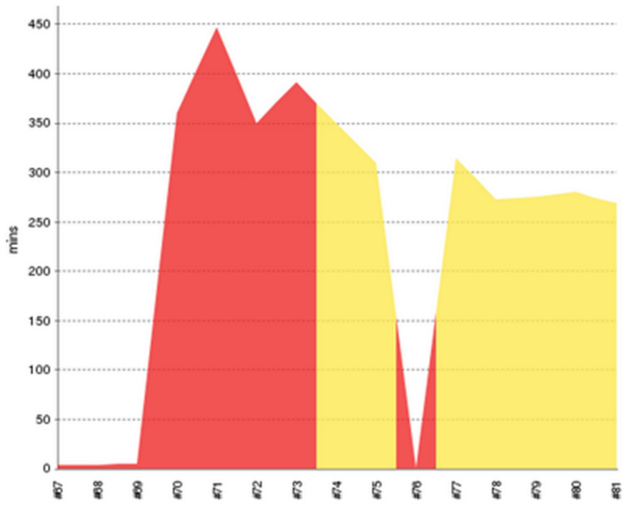


Figura 5.3. Desenvolupament del temps necessari per executar el TTE_COMMON[30]

Aquesta faceta no era un punt crucial de cara a l'execució en global de totes les classes sota test, com ja s'ha dit anteriorment les execucions tenen lloc a la nit fet pel qual una disminució de temps en aquesta tasca no suposa cap canvi per l'equip.

En el moment que era clau aquesta millora era a l'hora de reobrir o actualitzar un entorn per tal de millorar la seva cobertura. La mitjana d'entorns reproduïts al dia és de 6, si es consumia en mitjana 45 minuts, trobar-se una reducció del 43% afecta molt positivament a l'eficiència del grup de tests fins a arribar a quasi la meitat del temps emprat anteriorment. És més, amb aquesta suposició no es té en compte que s'hagi de fer una actualització de l'entorn (que equival al temps de reproducció des de 0 d'un entorn). Per tant, queda patent que l'esforç dedicat a aquesta millora sigut de gran ajuda.

[30] Imatge pròpia

Conclusions

Gràcies a les eines utilitzades a aquest projecte, s'ha aconseguit arribar a una cobertura pràcticament màxima (96% de línies de codi i 94% de branques condicionals) amb més facilitat per pensar els casos i els inputs com a conseqüència d'aquesta configuració. És més, s'ha reduït considerablement el temps que es necessitava per reproduir o actualitzar un entorn de test al mòdul més costós (3 hores menys en total). Un resultat molt satisfactori, tenint en compte que és la primera experiència amb VectorCAST i amb el C++.

Valorant el temps que es dedica a la tasca dels tests, em trobo amb pensaments contraposats i que són de vital importància. En primera instància, tot el treball i temps que t'estalvia el VectorCAST amb l'automatització i l'ajuda gràfica que proporciona ho contraresta amb l'esforç d'aprenentatge i la capacitat de reacció als seus errors. També és veritat que a llarg termini, aquest aspecte deixa de ser un problema. No obstant això, és un aspecte a tenir present, que potser fa que una empresa decideixi que és millor contractar a un enginyer amb experiència treballant amb l'eina si el projecte demana una certa pressa.

A partir d'aquesta experiència s'ha pogut construir un entorn que simula la part tractada en el projecte en petita escala. D'on s'han pogut extreure més solucions relacionades amb errors del VectorCAST i el Jenkins.

Respecte al C++, tot i compartir moltes semblances amb el C tractat a la meva enginyeria, ha sigut un repte fer el canvi de mentalitat per incloure totes aquestes eines i estructures d'alt nivell al coneixement previ. A part, també suposa una gran diferència fer els tests de codi que no és propi, una vivència que agraeixo haver tingut.

Malgrat estar satisfet amb el projecte és digne de tenir en compte la decepció de no poder mostrar la complexitat del codi tractat a causa de la confidencialitat del projecte de GTD. Tot i això, crec que he pogut desenvolupar uns entorns flexibles als canvis i robustos contra possibles errors.

Sense cap dubte, la integració contínua, que facilita un punt de connexió entre tantes disciplines del desenvolupament de software és una metodologia a seguir en un projecte. Valorant sobretot la part que he tractat, la dels tests, permet que els errors es detectin més fàcilment i abans que siguin potencialment irreversibles. Això comporta un estalvi de temps i pressupost del projecte considerable. A més, concentra la documentació per compartir tant amb l'equip com amb el client.

Bibliografía

Collins-Susman, Ben, W. Fitzpatric, Brian i Michael Pilato, Ben. 2006. Control de versiones Subversion. [En línia] 2006. http://svnbook.red-bean.com/en/1.2/svn-book.pdf.
Jenkins. 2018. Jenkins User Documentation. [En línia] Jenkins, 2018. https://jenkins.io/doc/.
JetBrains. 2018. TeamCity Documentation. [En línia] JetBrains, 2018. https://confluence.jetbrains.com/display/TCD18/TeamCity+Documentation.
Kirch-Prinz, Ulla i Prinz, Peter. 2002. A Complete Guide to Programming C++. [En línia] 2002. http://www.lmpt.univ-tours.fr/~volkov/C++.pdf.
Main, Chris. 2010. C++ Unit Test Frameworks. [En línia] Abril / 2010. https://accu.org/index.php/journals/1326.
McConell, Steve. 2004. *Code Complete.* Redmont : Pearson Education, 2004.
Noel. 2010. Exploring the C++ Unit Testing Framework Jungle. [En línia] 2010. http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle.
Osherove, Roy. 2013. *The Art Of Unit Testing With Examples In C#.* s.l. : Manning, 2013. 9781617290893.
TIOBE. 2018. [En línia] TIOBE, 2018. https://www.tiobe.com/tiobe-index/.
VectorCAST. 2017. VectorCAST documentation. [En línea] 2017. https://www.vectorcast.com/sites/default/files/downloads/pdf/vecchelp.pdf.
—. 2018. *VectorCAST Training 2018.* s.l. : VectorCAST, 2018. Font en pdf només accessible mitjançant el curs..
2017. Wikipedia. *DO-178B.* [En línia] 2017. https://es.wikipedia.org/wiki/DO-178B.

Annex 1. Descripció de visualitzacions al Jenkins

Secció A. Vista general dels Jobs

Com s'ha esmentat al projecte, posicionant-se a sobre de les icones es pot veure un breu resum de l'última execució duta a terme al Job (a la Figura A.1).

L'esfera canvia entre tres colors:

- Blau, si tot ha funcionat com s'esperava.
- Groc, si hi ha algun avís (warning) o un error menor concentrat a una sola classe.
- Vermell, si l'execució (build) no ha pogut acabar correctament.

En el cas del símbol del temps canvien des de tempesta fins al sol de la següent manera:

- El nombre de tests que s'executen correctament.
- La cobertura general que tenen els tests (línies de codi i branques condicionals).
- Estabilitat del Job, quants intents seguits porta funcionant correctament o incorrectament.

L'exigència d'aquests paràmetres, tal com s'ha explicat en el punt 3.4.2, es pot modificar a la configuració del Job en qualsevol moment amb els deguts permisos.

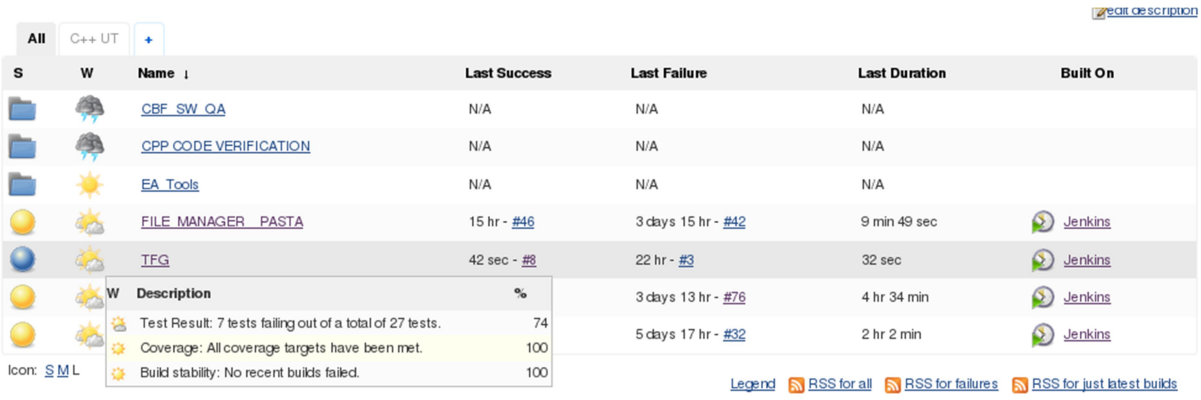


Figura A.1. Mostra bàsica de dades d'un Job[31]

[31] Imatge pròpia

Secció B. Vista dins del Job

Des d'aquí es té accés a tota la documentació de l'execució d'acord amb la Figura B.1:

- Des dels enllaços del centre es pot descarregar els arxius HTML generats per cada test i en global.
- A la columna dreta, les dues gràfiques mostrant la cobertura respecte a cada build i la complexitat o línies que conté el conjunt de tests.
- A l'esquerra, un menú amb cadascuna de les execucions mostra una per una les taules, sortides de la terminal que executa els entorns i gràfiques de cada intent.

Tota la informació esmentada en aquests punts és disponible per descarregar-se en qualsevol moment des del navegador web.

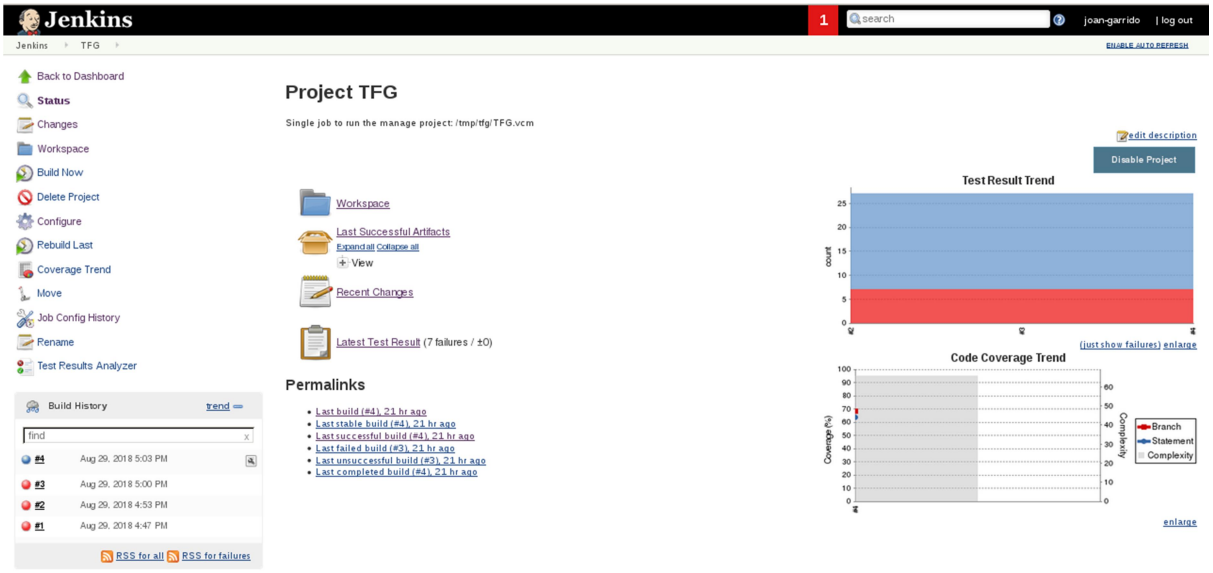
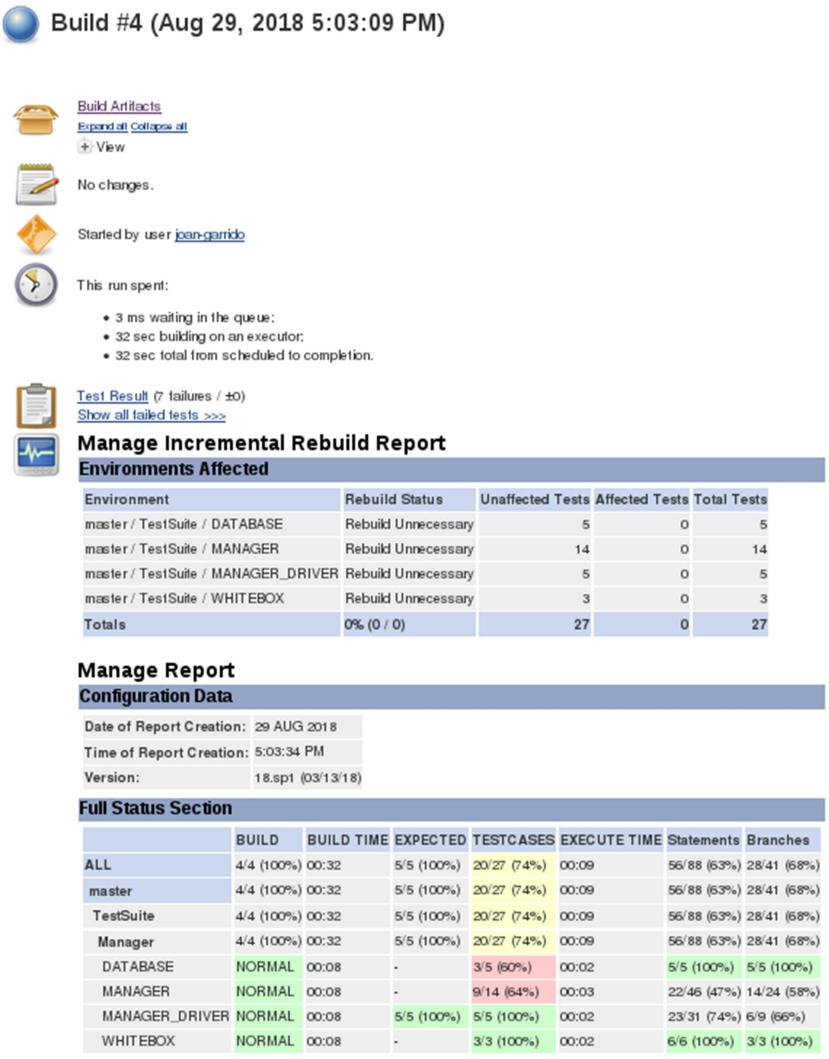


Figura B.1. Informació del menú d'un Job concret[32]

A aquesta Figura s'ensenya els detalls d'un build en general. Seguidament, un exemple de sortida de la consola i de gràfiques diverses:

- La de la Figura B.2, és la més semblant a un report complet d'una execució, aquest també pot ser accedit dins de la carpeta tst.
- En el cas de la Figura B.3 s'aconsegueix veure en forma de terminal cada pas que es fa durant l'execució i és on es mostren els errors més explícitament.
- Amb les Figures B.4 i B.5 es pot interactuar des del navegador, marcant i desmarcant les caselles es mostren les estadístiques dels conjunts volguts en qualsevol moment.

[32] Imatge pròpia



Build #4 (Aug 29, 2018 5:03:09 PM)

Build Artifacts
Expand all Collapse all
View

No changes.

Started by user joan-garrido

This run spent:

- 3 ms waiting in the queue;
- 32 sec building on an executor;
- 32 sec total from scheduled to completion.

Test Result (7 failures / ±0)
Show all failed tests >>>

Manage Incremental Rebuild Report

Environments Affected

Environment	Rebuild Status	Unaffected Tests	Affected Tests	Total Tests
master / TestSuite / DATABASE	Rebuild Unnecessary	5	0	5
master / TestSuite / MANAGER	Rebuild Unnecessary	14	0	14
master / TestSuite / MANAGER_DRIVER	Rebuild Unnecessary	5	0	5
master / TestSuite / WHITEBOX	Rebuild Unnecessary	3	0	3
Totals	0% (0 / 0)	27	0	27

Manage Report

Configuration Data

Date of Report Creation:	29 AUG 2018
Time of Report Creation:	5:03:34 PM
Version:	18.sp1 (03/13/18)

Full Status Section

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME	Statements	Branches
ALL	4/4 (100%)	00:32	5/5 (100%)	20/27 (74%)	00:09	56/88 (63%)	28/41 (68%)
master	4/4 (100%)	00:32	5/5 (100%)	20/27 (74%)	00:09	56/88 (63%)	28/41 (68%)
TestSuite	4/4 (100%)	00:32	5/5 (100%)	20/27 (74%)	00:09	56/88 (63%)	28/41 (68%)
Manager	4/4 (100%)	00:32	5/5 (100%)	20/27 (74%)	00:09	56/88 (63%)	28/41 (68%)
DATABASE	NORMAL	00:08	-	3/5 (60%)	00:02	5/5 (100%)	5/5 (100%)
MANAGER	NORMAL	00:08	-	9/14 (64%)	00:03	22/46 (47%)	14/24 (58%)
MANAGER_DRIVER	NORMAL	00:08	5/5 (100%)	5/5 (100%)	00:02	23/31 (74%)	6/9 (66%)
WHITEBOX	NORMAL	00:08	-	3/3 (100%)	00:02	6/6 (100%)	3/3 (100%)

Figura B.2. Report amb la informació de cada classe dels tests[33]

[33] Imatge pròpia

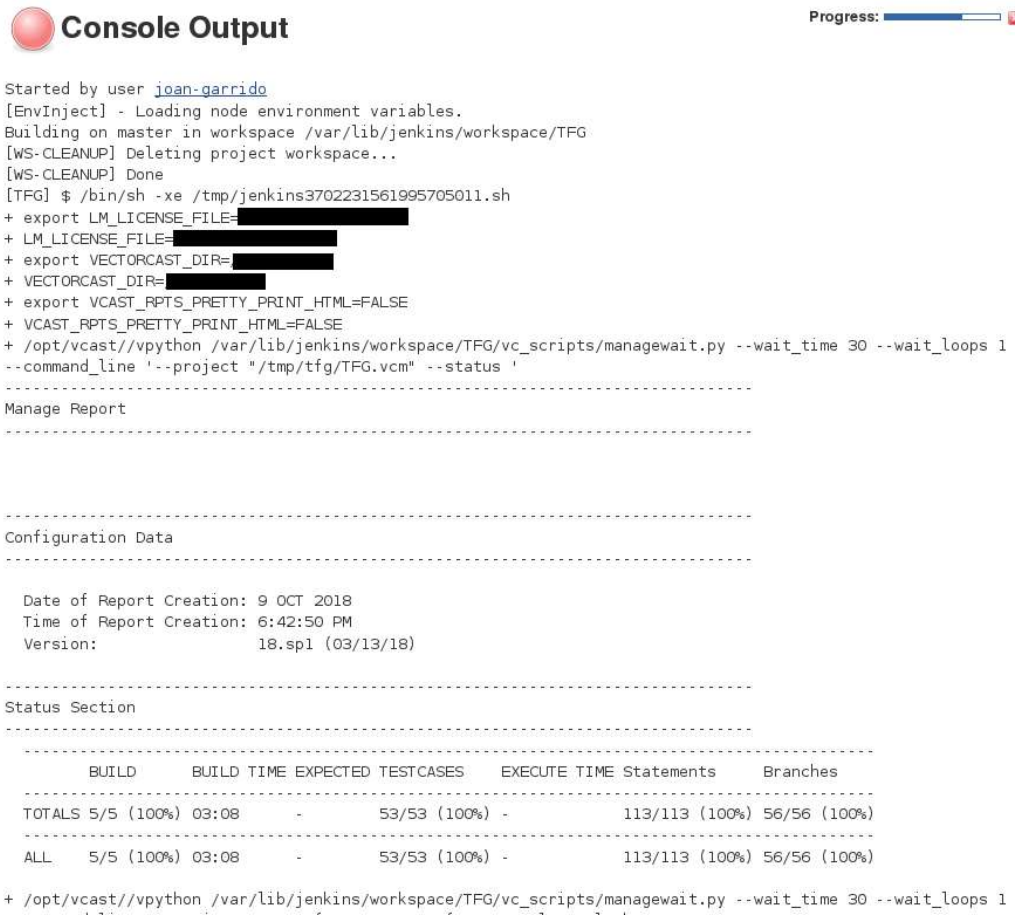


Figura B.3. Consola amb cada comanda que fa el VectorCAST[34]

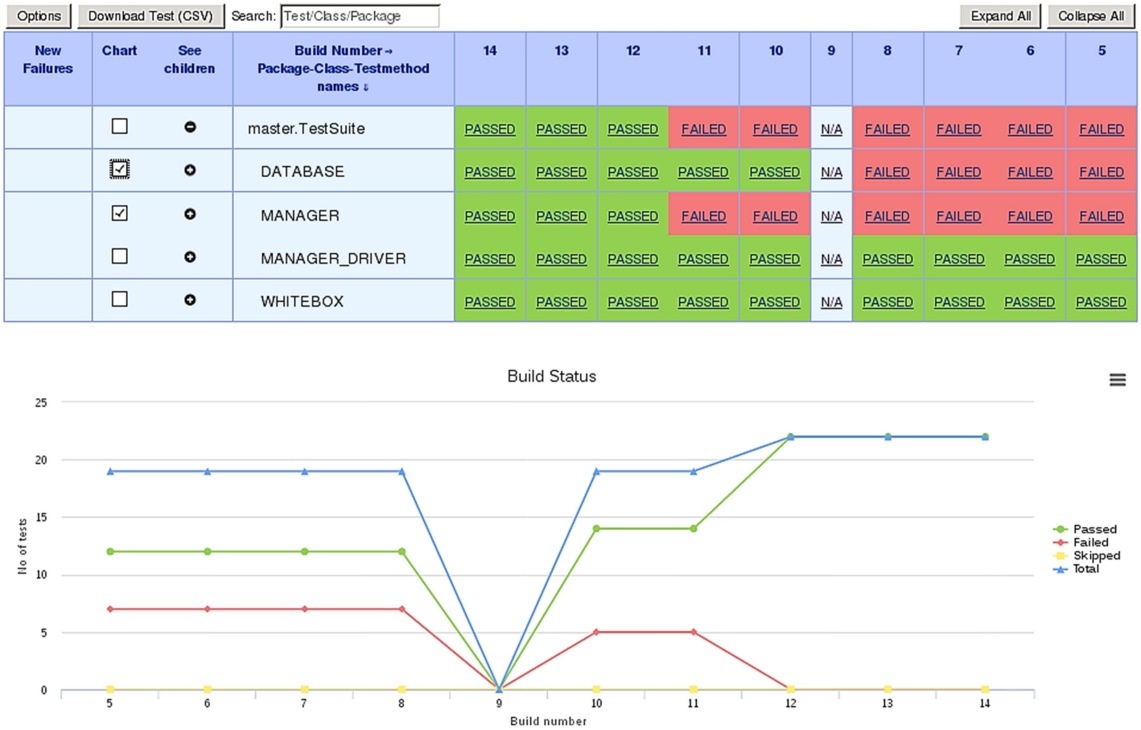


Figura B.4. Gràfica interactiva número 1[35]

[34] Imatge pròpia

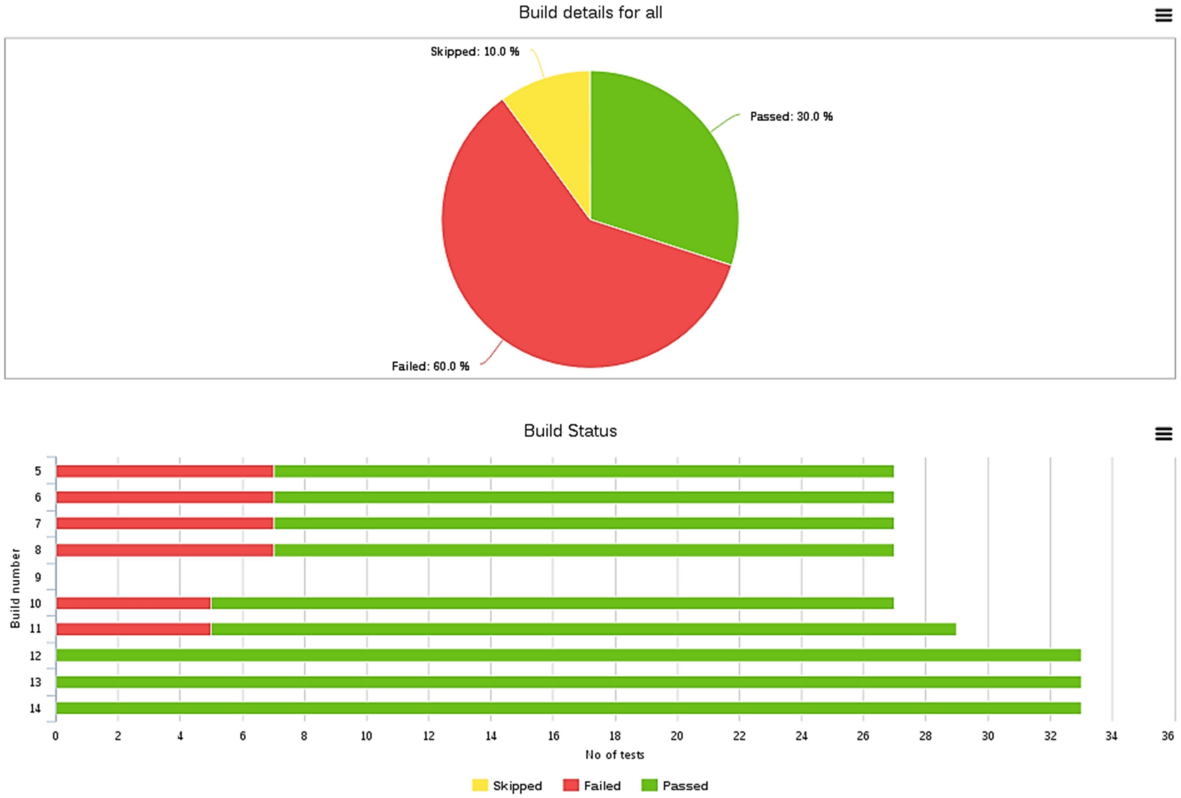


Figura B.5. Gràfica interactiva número 2[36]

[35] Imatge pròpia
[36] Imatge pròpia