Acte d'investidura de la Sra. Margaret H. Hamilton com a doctora *honoris causa* de la Universitat Politècnica de Catalunya·BarcelonaTech

18 d'octubre de 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Discurs pronunciat per la nova doctora *honoris causa*, Sra. Margaret H. Hamilton

Un paradigma preventiu per a *software* i sistemes

La meva formació en *software* (programari i tot allò relatiu a la seva construcció i manteniment) no ha estat gens tradicional. Va començar abans que la disciplina de dissenyar i desenvolupar *software* fos un camp d'estudi. No hi havia cursos disponibles per entendre què fèiem o com ho havíem de fer. Tota l'educació anterior, per dir-ne d'alguna manera, era una combinació de l'aprenentatge d'experiències vitals i laborals aparentment no relacionades de l'època preuniversitària i universitària, en què em vaig especialitzar en matemàtiques i en filosofia com a mínor. Durant anys vaig treballar intensament en aplicacions múltiples i variades que demanaven fer coses que no s'havien fet abans, i al mateix temps vaig haver de trobar solucions i normes i tècniques associades per poder fer coses semblants en el futur. Mirant-ho en perspectiva, crec que tot va començar quan era ben jove. Recordo, de petita, la pila de xerrades filosòfiques del tipus «i si...?», «per què?» i «per què no?» que tenia amb el meu pare i el meu avi. Tots dos em van ensenyar a observar i a qüestionar-m'ho tot fins que les respostes tinguessin sentit. I mirant enrere, veig com n'eren d'importants aquestes xerrades per al que seria important per a la meva feina en el futur.

Vaig tenir la sort de tenir reptes, responsabilitats i experiències laborals excepcionals, fins i tot abans d'anar a l'institut. I tot perquè em calia estalviar per a la universitat. Una experiència concreta va ser rellevant. Abans de fer els 15 anys, em va contractar el propietari d'una mina de coure abandonada, a la península superior de Michigan, que havia estat reconvertida en un magatzem de plàtans. Ell volia transformar la mina en una atracció turística on la gent fes visites guiades per la mina i els expliquessin com s'extreia el coure natiu (era l'únic lloc del món on existia el coure natiu). Volia que l'ajudés a arrencar aquest negoci turístic nou. Ell no havia passat de quart de primària i sabia que no tenia prou base (com ara matemàtiques) per fer-ho tot ell. Em va demanar que aprengués tot el que pogués sobre el coure, que fos la primera guia de la mina i que contractés i formés persones que treballessin per a mi com a guies. En el nostre negoci turístic vam passar de fer la visita a dues persones al dia a fer-la a milers de persones. Jo no solament em feia càrrec dels guies, sinó que també era responsable de gairebé tot el que estigués relacionat amb la mina, com ara engegar i gestionar una botiga de regals, fer-me càrrec dels comptes i portar el negoci en general. En aquell temps vaig aprendre aviat que, si cometies errors, no et podies permetre a tu mateixa (ni als altres!) tornar-los a come-

tre. Després d'aquesta experiència, quan més endavant va arribar l'hora d'aprendre a fer altres tipus de feines completament noves o desconegudes sense cap mena de coneixement previ ni experiència i responsabilitzar-me'n, no va resultar cap problema: o almenys era el que em pensava!

També m'adono de la sort que vaig tenir d'haver passat per tantes experiències increïbles durant els primers temps del *software*, quan érem ben bé al peu del canó, perquè érem en un camp d'estudi abans que ho fos i vaig aprendre molt dels companys. El 1959, Edward N. Lorenz, un professor del MIT, em va introduir en la informàtica; hi vaig desenvolupar aplicacions de predicció del temps en hexadecimal i binari en l'ordinador LGP-30. Conegut pel seu treball capdavanter en la teoria del caos, era una persona molt humil. El seu amor per l'experimentació en *software* era contagiós, i em va contagiar aquest virus. Amb aquesta finalitat, era crucial entendre les relacions entre el *hardware* (maquinari i tot allò relatiu a la seva construcció i manteniment) i el *software*, perquè en aquest projecte era important desenvolupar un *software* optimitzat per al rendiment. Aleshores vèiem els errors només com una nosa, sobretot perquè les sessions de depuració no s'acabaven mai.

Un altre dels primers projectes va ser el SAGE (Semi-Automatic Ground Environment), en què vaig desenvolupar aplicacions en llenguatge d'assemblador en el primer AN/FSQ-7 (l'XD-1), als laboratoris Lincoln del MIT, per buscar avions enemics. La màquina era enorme, era el sistema informàtic més gran que s'havia muntat mai. Era especialment important no cometre cap error perquè, si ho feies, l'ordinador et delatava amb unes sirenes i botzines i uns flaixos que tothom veia i sentia, dins d'un edifici molt gran que semblava un magatzem. Els operadors i els programadors venien corrents per saber de qui era el programa que s'havia penjat. Una vegada, a mitja nit, em va trucar un dels operadors informàtics tot esverat dient-me que al meu programa li passava alguna cosa seriosa: ja no feia el so de les onades a la platja que solia fer. Aleshores va quedar clar que havíem trobat una nova manera de depurar, fent servir el so!

Tal com passava amb l'LGP-30, depurar en l'AN/FSQ-7 del projecte SAGE prenia molt temps, i no existien eines per trobar un error i el que el provocava. Els errors de *software* eren mal rebuts especialment perquè es consideraven una molèstia (o una vergonya). Durant aquest projecte, com en el projecte LGP-30, em fascinaven els errors: buscàvem maneres d'entendre què feia que succeïssin un o diversos errors específics, o uns tipus d'errors, i desenvolupàvem maneres d'evitar que succeïssin en el futur.

Aleshores, estaves tota sola i el coneixement (o la manca d'aquest coneixement) es transmetia de persona a persona. Un director et contractava si coneixies les ordres del llenguatge natiu del seu ordinador. Com si conèixer un conjunt de paraules en anglès significués que pots escriure una novel·la en aquest idioma. Aquesta actitud em deixava perplexa. Si un sistema es bloquejava, sempre en tenia la culpa el *software*. Els termes no estaven definits, i això comportava errors, malentesos i drames. Termes com *software, disseny, error* i *sistemes informàtics* significaven coses diferents per a persones diferents. De fet, un dels directors per a qui vaig treballar al començament es pensava que *software* volia dir «soft wear», és a dir, «roba lleugera». A més, en aquell temps, el món del *software* era una minoria. Els informàtics de l'època sovint podien embolicar-ho tot en el cas que desconeguessin altres camps de *software* (com ara, barrejar la funcionalitat del sistema operatiu de base amb la funcionalitat del sistema objecte de l'aplicació). Encara que aleshores algunes coses eren força diferents, el procés del cicle vital en si mateix, en general, no diferia gaire del cicle vital típic d'avui (per exemple, els models de desenvolupament en cascada, en espiral i àgil), anava dels requisits a la codificació, passant per una verificació i un manteniment interminables.

El *software* de navegació de l'Apollo

El SAGE va tenir molts problemes, que estaven molt relacionats amb els errors, però això només era el començament del que vindria després: el projecte del *software* de navegació de l'Apollo fet al MIT, mitjançant un contracte amb la NASA. El desafiament

era que el *software* estaria pensat per a persones. Això volia dir que els astronautes s'hi jugaven la vida. Havia de FUNCIONAR a la primera. El *software* no només havia de ser ultrasegur, sinó també que havia de ser capaç de detectar un error i recuperar-se'n en temps real. L'aprenentatge es produïa treballant-hi. Els enginyers de hardware van trobar normes formals per dissenyar i construir aquest hardware, però nosaltres, els «enginyers del *software*», no ho podíem fer així. S'havien de resoldre problemes que no s'havien resolt mai. De vegades ens ho inventàvem. La majoria dels desenvolupadors eren intrèpids i joves i, tot i així, la dedicació i el compromís es donaven per fets. Els directors (procedents sobretot de l'àrea de *hardware*), per als quals el *software* era un misteri, ens van donar llibertat i confiança totals. No teníem temps de ser aprenents. Vaig començar creant *software* per a les missions no tripulades i em vaig concentrar en les àrees de *software* de sistema, que eren utilitzades per tot el *software* de navegació i que l'afectaven. El *software* de navegació incloïa el *software* per a la detecció d'errors i la recuperació. El sistema de *software* per a les missions no tripulades era síncron.

Després van venir les missions tripulades. Jo em feia càrrec de l'equip que desenvolupava el *software* de navegació per a les missions tripulades i del *software* de navegació en si mateix, però per norma general em vaig assegurar d'estar al dia tècnicament de l'àrea del *software* de sistemes. El *software* per a les missions tripulades era més complex. En el nostre entorn era asíncron (un entorn de programació múltiple), en què tasques d'alta prioritat interrompien tasques de prioritat més baixa, basant-se en la prioritat de cada tasca en relació amb la prioritat de la resta de tasques. Ens corresponia a nosaltres, els desenvolupadors, assignar manualment una prioritat única a cada procés del *software* de navegació per estar segurs que tots els esdeveniments tindrien lloc en l'ordre correcte i a l'hora prevista. Jo sempre buscava noves maneres perquè es preparés per als imprevistos i se'n recuperés: des de la protecció específica de cada programa a la protecció del conjunt del sistema. Vaig començar a pensar en tots els possibles «i si...?», i vaig treballar en maneres d'afrontar-los. Per exemple, em vaig preguntar: I si hi hagués una emergència durant el vol i volguéssim avisar els astronautes? Vaig pensar que havia d'haver-hi una manera de solucionar-ho. Sí que hi era!

Totes les missions eren emocionants, però l'Apollo 11 era especial. Mai no havíem aterrat a la Lluna. Tot anava a la perfecció fins que va passar una cosa del tot imprevista. Just quan els astronautes es disposaven a aterrar a la Lluna, l'ordinador de vol es va sobrecarregar! Les visualitzacions de prioritat del *software* (conegudes també com *display interface routines*) de les alarmes 1201 i 1202 van interrompre les pantalles normals de la missió dels astronautes per avisar-los d'una emergència; van permetre que el control de la missió de la NASA entengués què passava i van alertar els astronautes perquè tornessin a posar el commutador del radar de retrobament en la posició correcta. Aviat va quedar clar que el *software* no només informava tothom que hi havia un problema relacionat amb el *hardware*, sinó que també el compensava. Les visualitzacions de prioritat van donar als astronautes una decisió del tipus endavant / no endavant (aterrar o no aterrar). En només uns minuts de temps es va prendre la decisió de tirar endavant l'aterratge. La resta ja és història. Els tripulants de l'Apollo 11 van ser els primers humans que van caminar per la Lluna i el nostre *software* es va convertir en el primer *software* que va funcionar a la Lluna.**[1,2,3]**

Els mecanismes de detecció i recuperació d'errors de *software* i sistema inclosos en el *software* havien pres el control. La sobrecàrrega va activar reinicis de captura-restitució del tipus «finalitza i recalcula» des d'un lloc segur en tot el sistema. Només es van mantenir les tasques amb prioritat més alta. Durant aquells moments, vaig recordar el descobriment més emocionant i memorable que hi estava relacionat. Va ser quan em vaig adonar que les passes que havíem fet anteriorment dins de l'entorn de multiprogramació podien convertir-se en la base per a solucions en un entorn de multiprocessament. És a dir, encara que només hi hagués un procés executant-se dins d'un entorn de multiprogramació, altres processos esperaven paral·lelament aquest procés. Amb aquest rerefons, es van poder crear les visualitzacions de prioritat i es va intercanviar la interfície entre el

software de navegació i els astronautes de síncron a asíncron (el *software* i els astronautes van esdevenir processos paral·lels dins un sistema de sistemes). Això no hauria estat possible sense una estratègia de sistema de sistemes (i equips) integrat i les aportacions fetes per altres grups per ajudar a fer-ho realitat. L'equip de *hardware* del MIT va canviar el seu *hardware* i l'equip de planificació de la missió a Houston va canviar els procediments dels astronautes; tots dos van treballar estretament amb nosaltres per adaptar les visualitzacions de prioritat tant del mòdul de comandament (MC) com del mòdul lunar (ML) per a tota mena d'emergències i al llarg de qualsevol missió.

Vaig pensar en els anys que ens vam estar preparant per a aquell dia i en la sort que tenia de treballar i compartir aquella experiència amb tantes persones amb talent i dedicació que ho van fer possible. Fer descobertes, pensar noves idees i trobar solucions va ser una aventura. Des del meu punt de vista, l'experiència del *software* mateix (dissenyar-lo, desenvolupar-lo, fer-lo evolucionar, veure'n el rendiment i aprendre'n per a sistemes futurs) va ser com a mínim tan emocionant com els esdeveniments relacionats amb la missió.

Com a desenvolupadors, se'ns havia donat una oportunitat única a la vida: cometre tots els tipus d'errors humanament possibles, cadascun dels quals amagava respostes a preguntes que no ens havíem fet. Veient-ho en perspectiva, d'un gran mal en surt un gran bé. La tasca en qüestió era desenvolupar el *software* del MC i del ML. Això incloïa el *software* del sistema que s'ubicava tant dins del MC com del ML i que tots dos compartien, i l'estructura del *software* («l'adhesiu») que descrivia les relacions que hi havia entre si, en mig de les fases de la missió i a dins de la missió. Les actualitzacions arribaven contínuament de centenars de persones (incloent-hi «convidats») al llarg del temps i de la multitud de versions per a cada missió (en què el *software* d'una missió es treballava paral·lelament amb el *software* d'altres missions). Ens vam haver d'assegurar que tot lligaria, que les parts del *software* interactuarien i funcionarien les unes amb les altres, i també amb altres sistemes (incloent-hi el *hardware*, i els recursos humans de la missió).

Estàvem limitats per restriccions d'espai i temps del *hardware*, i això donava als «experts» de *software* llicència per ser creatius. El codi «creatiu» (és a dir, la programació enginyosa) era més apreciat que la quantitat de línies de codi que pogués escriure una persona. Els requisits eren descartats pels experts aliens al *software*, que donaven per fet que, d'alguna manera, tots els programes interactuarien junts per art de màgia. Per sort, no va ser el cas. I és que, si hagués estat així, no hauríem après mai què podia passar després. Encara més, a causa de les limitacions de l'ordinador, les ubicacions d'emmagatzematge de les dades es compartien entre les fases de la missió, al llarg de les fases i entre les fases, i, a causa de l'entorn de multiprogramació, les responsabilitats i les ubicacions d'emmagatzematge de dades també eren compartides entre molts programes que interrompien contínuament programes de prioritat més baixa basant-se en l'hora i la prioritat durant cada una de les fases de la missió. Això incloïa multiprocessament síncron i asíncron amb intervenció humana dins d'un sistema de sistemes. Tot i que hi havia moltes possibilitats per cometre errors, aleshores hi havia possibilitats de trobar solucions per evitar-los. Les regles de l'enginyeria del *software* evolucionaven amb cada descobriment rellevant, mentre que les normes de l'alta direcció de la NASA van passar de la llibertat total a l'exageració. *Encara que no van faltar possibilitats de cometre un error i gairebé qualsevol mena d'error possible, pel que sabem, mai no hi va haver cap error de navegació durant el vol.*

Quan un passa per aquestes experiències, no pot evitar intentar aprendre'n. Vam preguntar-nos: Què podem fer millor per a sistemes futurs? Com que ho estem fent bé, què hem de continuar fent? Amb el finançament inicial de la NASA i el Departament de Defensa, vam dur a terme un estudi empíric del treball de l'Apollo. La nostra anàlisi va abordar molts aspectes, no només per a missions espacials, sinó també per a sistemes en general. Les lliçons apreses d'aquest esforç (i el seu impacte) continuen

avui: demana't sempre: «I si...?», i sempre espera l'inesperat. Vam aprendre que els sistemes són asíncrons, distribuïts i dirigits per esdeveniments en la natura i que això s'havia de reflectir en el llenguatge per definir-los i en les eines per crear-los, caracteritzant-ne el comportament natural en termes de semàntica d'execució en temps real. Havent fet això, ja no cal definir explícitament planificacions sobre quan tenen lloc els esdeveniments. Descrivint les interaccions entre els objectes, la planificació d'esdeveniments es defineix per si mateixa. També vam aprendre que el cicle vital d'un sistema objecte és un sistema amb un cicle vital propi i que cada sistema és, intrínsecament, un sistema de sistemes.

El més interessant de tot van ser les lliçons que vam aprendre dels errors detectats durant les verificacions anteriors al vol. Estaven plens de sorpreses. Ens van dir què fer i on anar. Després de categoritzar els errors, vam concentrar esforços a analitzar tres categories d'errors: errors d'interfície (conflictes de dades, temporals i de prioritat), que eren el 75 % de tots els errors detectats; errors detectats de manera manual amb l'Augekugel (per observació) o el mètode d'escanejar codi imprès, i errors previs que existien en missions anteriors, que normalment eren els errors més subtils i els més difícils de trobar. Les nostres anàlisis van tenir com a conseqüència una teoria basada en les lliçons apreses dels projectes Apollo i posteriors. Dels seus axiomes, en vam extraure un conjunt de patrons considerables per definir un sistema. Això va dur a un llenguatge de sistemes universal (USL), juntament amb la seva automatització i el paradigma de desenvolupament preventiu, el desenvolupament abans dels fets (DBTF).**[2,4,5,6,7,8,9]**

Va arribar un moment en què va quedar clar que els sistemes definits amb l'USL es comportaven de manera diferent d'aquells que estaven definits amb els llenguatges tradicionals. Va ser molt interessant el descobriment que el problema d'arrel de l'enginyeria de sistemes tradicional i dels llenguatges de desenvolupament de *software* i els seus entorns és que ajuden els usuaris a arreglar els errors (després dels fets) en comptes de fer les coses bé des del començament (abans dels fets). En contrast amb això, amb un paradigma preventiu, en lloc de buscar més maneres de buscar errors i continuar buscant errors al final del cicle vital, la majoria d'errors, incloent-hi tots els errors d'interfície, ja des del començament no tenen cabuda en un sistema per la manera com s'ha concebut. Buscar errors inexistents amb verificacions es converteix en un esforç obsolet.

Seguim descobrint noves propietats en sistemes concebuts amb l'USL. Cada propietat s'hi afegeix ella sola al llarg del desenvolupament propi d'un sistema, els derivats de la definició del sistema (el seu *software* n'és un) hereten les propietats de la definició de la qual deriven, la integració del *software* a partir dels sistemes és perfecta, la reutilització és inherent o derivable; un sistema concebut amb aquest llenguatge té propietats en la seva definició que intrínsecament ajuden al seu propi desenvolupament, abans dels fets, i **[2]** cadascun dels seus sistemes té la fiabilitat i la productivitat incorporades al llarg del seu cicle vital. Al contrari que el paradigma tradicional i la seva filosofia de verificar fins a l'extenuació, amb el paradigma preventiu, com més fiable és el sistema, més gran és la productivitat en construir-lo, i es minimitza la necessitat de la majoria de verificacions. Gran part del disseny i la totalitat del codi estan generats per l'automatització de l'USL, i hereten totes les propietats de la definició de la qual han sorgit. El desenvolupador no ha de codificar mai a mà ni canviar el codi a mà; només es regenera la part del sistema canviada i s'integra automàticament amb la resta de l'aplicació.

Vam aprendre que, com que l'USL és independent en sintaxi, en implementació i en arquitectura, els seus sistemes es poden desenvolupar per a arquitectures diverses, i que la sintaxi d'altres llenguatges es pot mapar en les semàntiques subjacents de l'USL.**[7]** Al contrari d'un llenguatge formal de base matemàtica però limitat quant a abast des d'un punt de vista pràctic (per exemple, el tipus o la mida del sistema), l'USL expandeix la matemàtica tradicional amb un concepte únic de control, i li permet

ser compatible amb la definició d'un sistema de qualsevol tipus o mida. Amb aquest paradigma, el llenguatge també contribueix a la seva automatització, i li serveix de base perquè hereti i transmeti les propietats de control de la definició per al mateix procés de desenvolupament del sistema. L'automatització de l'USL, un sistema gran per si mateix (milions de línies de codi), s'autodefineix i s'autogenera.

No és màgia. Aquestes coses tan sols són possibles a causa de la base matemàtica de l'USL. No només té les seves arrels en el sistema de l'Apollo i posteriors, també té arrels en altres mètodes formals, lingüística formal i tecnologies d'objectes. Ha evolucionat durant dècades i sempre ha estat autosuficient en tipus diferents d'entorns, incloent-hi l'acadèmic,**[6,10,11]** agències governamentals,**[12]** el comercial**[8]** i altres entorns.**[13]** Utilitzat en recerca i per organitzacions pioneres, s'ha posicionat per un ús més extens. Separant-se radicalment del que és tradicional, redefineix el que és possible. Com que és nou en el món en general, seria natural fer suposicions sobre què és possible i impossible basant-nos en la seva similitud superficial amb altres llenguatges, com ara els llenguatges tradicionals orientats a objectes. Hem après que és útil posar en suspensió totes les nocions preconcebudes quan un s'introdueix en aquest llenguatge, perquè és un món en si mateix, una forma diferent de pensar en sistemes.

Quan als desenvolupadors de sistemes mitjans o grans d'avui en dia se'ls demana que facin una llista dels seus problemes més recurrents, diuen: la integració, si és possible, s'endarrereix massa; la traçabilitat, la flexibilitat i la capacitat d'evolucionar no hi són; la reutilització és *ad hoc* i tendeix als errors; el *software* no és fiable ni tan sols amb una verificació intensiva; el *software* costa massa i triga massa a crear-se. Sens dubte, els últims problemes de la llista que tenen a veure amb fiabilitat i productivitat queden resolts si la majoria dels altres problemes de la llista, si no tots, es resolen. La majoria de la gent diu que no és possible fer gairebé res per abordar aquests problemes, almenys en un futur previsible; el *software*, per la seva naturalesa, està destinat a generar aquesta mena de problemes. Jo pregunto: com és que els desenvolupadors d'avui esmenten els mateixos problemes que esmentaven fa 50 anys quan era un camp d'estudi acabat de néixer, i com és que hi donen la mateixa explicació que hi donaven fa 50 anys?

És cert que molts dels problemes de *software* recurrents que hi havia al principi avui encara hi són. Tanmateix, el que hem après, de la nostra pròpia feina, és que l'explicació que hi donaven aleshores i també ara no és la que jo hi donaria. És a dir, jo crec que la raó dels problemes relacionats amb el desenvolupament de *software* no és la naturalesa del *software per se*, sinó que en gran mesura tenen lloc a causa del paradigma tradicional que s'ha utilitzat per crear-lo, un paradigma que ha estat present des del començament i continua en vigor fins a l'actualitat. Moltes de les dificultats conegudes per tothom amb el paradigma tradicional ja no existeixen amb un paradigma preventiu. A més a més, el que funciona millor per desenvolupar sistemes ultrafiables resulta que funciona millor per a tots els sistemes en general, independentment de l'aplicació. S'ha demostrat que molts aspectes dels problemes recurrents (de fa 50 anys i d'avui) poden ser abordats, per no dir eliminats del tot definitivament, utilitzant el paradigma preventiu.**[2,4,7,8,9,10,11,12,13]** El paradigma preventiu, amb el seu llenguatge i la seva automatització, no ha decebut quan s'ha posat a prova. De fet, com més gran i complex és el sistema, millors són els resultats. El que pugui fer-se amb el que s'ha après, tanmateix, depèn de com de disposats i d'oberts estiguin els desenvolupadors envers un canvi sobre com es crea el *software*. Tenint en compte els tipus de sistemes que necessitem per construir avui i demà, crec que aquest canvi ha de començar i començarà a produir-se aviat.

Tots els meus èxits, en gran part s'han produït perquè he estat al lloc adient en el moment oportú, amb les oportunitats i la gent adequades. D'alguna manera, vaig tenir l'avantatge de començar sense cap noció preconcebuda, ja que calia endinsar-se

en camps que mai s'havien explorat abans. Moltes de les coses que fèiem no s'havien fet abans i això em fa sentir molt afortunada. El mèrit no només és de les persones de qui he après tant, sinó també del fet que he comès errors en els quals he tingut la sort d'haver tingut alguna responsabilitat. Sense aquests errors no hauríem pogut aprendre les coses que vam aprendre. Alguns d'escandalosos i sovint prou coneguts per no voler que tornin a passar mai!

Els errors ens han ensenyat com viure sense aquests. Ens han dut a un llenguatge amb un paradigma preventiu en què la definició d'un sistema *substitueix intrínsecament* molt del que abans eren aspectes del mateix cicle vital del sistema, i que ara ja no es necessiten, i serveix *d'aportació* a l'automatització del que abans eren processos manuals en el cicle vital del sistema. Per tant, fa que ja no calguin moltes parts del propi cicle vital del sistema. En resum, explorar i arriscar-nos en territori desconegut ens ha conduït, entre altres coses, als errors. Els errors ens han conduït a un paradigma que ens porta abans dels fets cap al futur. Ensenyar la gent com pensar, fer i ser des del punt de vista del paradigma és el veritable repte que ens espera.

Bibliografia

[1] Snyder, L and Henry, RL. «Fluency7 with Information Technology». Pearson, New York, NY; 2018, p. 173-176.

[2] Hamilton, MH. «The Language as a Software Engineer». 2018 International Conference on Software Engineering; 27 May - 3 June, 2018; Gothenburg, Sweden. Celebrating its 40th anniversary, and 50 years of Software engineering.

[3] Hamilton MH. «Computer Got Loaded». Letter to Datamation. Cahners Publishing Company; 1 March 1971.

[4] Hamilton, MH and Hackler, WR. «Universal Systems Language: Lessons Learned from Apollo». IEEE Computer. doi:10.1109/MC.2008.541. December 2008.

[5] Hamilton Technologies, Inc. 001 Tool Suite (1986-2018). Example demo: «System: do_all_taxes». Cambridge, MA; 25 Aug. 2018.

[6] Hamilton, MH and Hackler, WR. «Universal Systems Language for Preventative Systems Engineering». Proc. 5th Ann. Conf. Systems Eng. Res. (CSER), paper #36; Stevens Institute of Technology; Mar. 2007.

[7] Hamilton, MH and Hackler, WR. «A Formal Universal Systems Semantics for SysML2». 17th Annual International Symposium, INCOSE 2007, paper #8.3.2; San Diego, CA; Jun. 2007.

[8] Hamilton, MH. «Universal Systems Language (USL) and its Automation, the 001 Tool Suite, for Designing and Building Systems and Software». IEEE Computer Society/Lockheed Martin Webinar Series, slides 36-41; 27 Sept. 2012.

[9] Hamilton, MH. «What the Errors Tell Us». IEEE Software-Special issue, «50 years of Software Engineering». September/October 2018; 35(5).

[10] Ouyang, M and Golay, MW. «An Integrated Formal Approach for Developing High Quality Software of Safety-Critical Systems». Report No. MIT-ANP-TR-035; Massachusetts Institute of Technology, Cambridge, MA; 1995.

[11] Krut Jr., B. «Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology». CMU/SEI-93-TR-11, ESC-TR-93-188. SEI; Carnegie Mellon University, Pittsburgh, PA; 1993.

[12] Department of Defense. «National Test Bed Software Engineering Tools Experiment —final report», vol. 1, Experiment Summary, Table 1, p. 9. DoD Strategic Defense Initiative Organization; Washington, DC; Oct. 1992.

[13] Schindler, MJ. «Computer-Aided Software Design: Building Quality Software with Case». John Wiley & Sons; New York, NY; 1990.