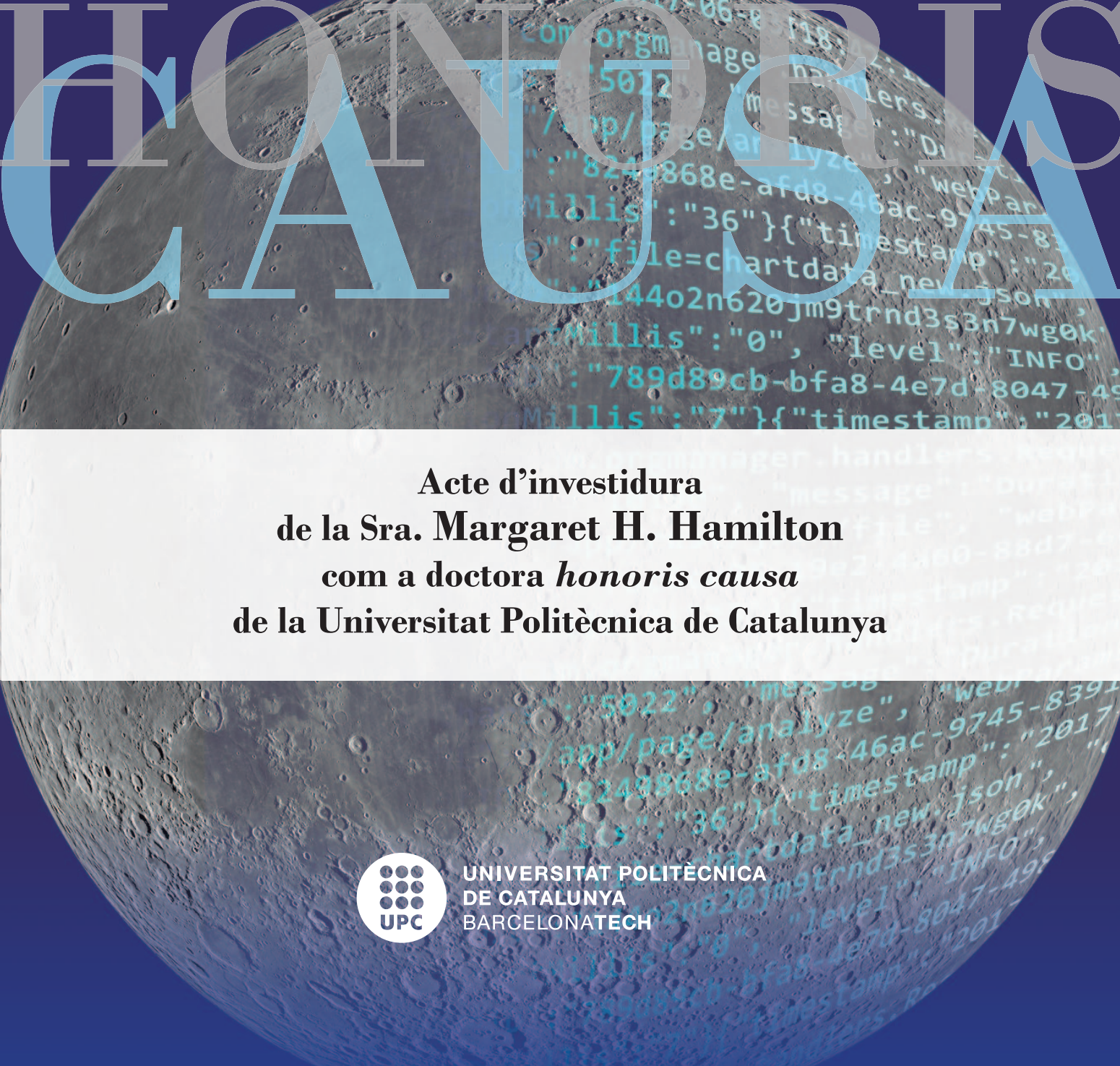

Acte d'investidura de la Sra. Margaret H. Hamilton com a doctora *honoris causa* de la Universitat Politècnica de Catalunya

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

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

Imprès en paper ecològic

Servei de Comunicació de la UPC, 2018 (9838)

Índex /
Table of contents

Ordre de l'acte d'investidura

Benvinguda del rector de la Universitat Politècnica de Catalunya · BarcelonaTech, Prof. Francesc Torres.

Lectura de l'acord del Consell de Govern, a càrrec de la secretària general, Sra. Marta de Blas.

Laudatio de la padrina, Prof. Núria Castell Ariño.

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

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

Paraules del rector, Prof. Francesc Torres.

Interpretacions musicals a càrrec de
la Coral Gaudeamus de l'EEBE,
la Coral del Parc Mediterrani de la Tecnologia,
el Cor Ol·lari de l'FME,
la Coral Arquitectura i l'Orquestra de la UPC:

Canticorum iubilo, Georg Friedrich Händel

Cançó de bres per a una princesa negra, A. Rodríguez Sabanes, harmonització de D. Antolí

A round of three country dances in one, harmonització de Thomas Ravenscroft

Gaudeamus igitur, harmonització de Joan Casulleres

Gaudeamus igitur

Gaudeamus igitur,
iuvenes dum sumus. (bis)
Post iucundam iuventutem,
post molestam senectutem,
nos habebit humus.

Vivat academia,
vivant professores.
Vivat membrum quodlibet,
vivant membra quaelibet,
semper sint in flore.

Gaudeamus igitur,
iuvenes dum sumus. (bis)
Post iucundam iuventutem,
post molestam senectutem,
nos habebit humus.

Elogi dels mèrits de la Sra. Margaret H. Hamilton

Prof. Núria Castell Ariño
Universitat Politècnica de Catalunya · BarcelonaTech

Distingit rector de la Universitat Politècnica de Catalunya · BarcelonaTech, distingits membres del Claustre Universitari i Consell Social, autoritats i representants d'institucions i empreses, professors, estudiants, personal administratiu, família i amics, i estimada Sra. Margaret H. Hamilton.

És un gran plaer ser aquí avui en aquesta cerimònia solemne per donar la benvinguda al Claustre Universitari a una nova doctora *honoris causa* de la Universitat Politècnica de Catalunya (UPC), d'acord amb la Resolució del Consell de Govern de 23 de maig de 2017, proposada per la Facultat d'Informàtica de Barcelona (FIB), de la qual em sento orgullosa d'haver estat degana. La nominació de Margaret H. Hamilton va obtenir el suport de departaments i escoles de la UPC i de diversos experts internacionals en enginyeria del *software* i història de la informàtica.

Margaret H. Hamilton té una llarga llista d'èxits. És informàtica, enginyera de sistemes i empresària. També és coneguda com la dona que va portar l'home a la Lluna. Durant la seva llarga carrera de recerca i professional, els seus interessos de recerca han inclòs els sistemes i el *software* amb paradigma preventiu, la teoria formal, els sistemes ultrafiables, la detecció i recuperació d'errors, els llenguatges de modelització i programació, la reusabilitat, la correcció per les propietats integrades del llenguatge, les interfícies home-màquina, les arquitectures obertes, la integració sense fissures, els sistemes operatius, els entorns automatitzats de cicle de vida, la productivitat en el desenvolupament del *software*, el disseny de sistemes i el desenvolupament de *software* dins d'un entorn de sistema de sistemes asíncron i distribuït, i la gestió del cicle de vida del *software*.

Hamilton va començar a treballar en diversos projectes de *software* al Massachusetts Institute of Technology (MIT) a Boston, on va aprendre sobre sistemes i *software* per ella mateixa. Aleshores va obtenir un càrrec a la NASA com a desenvolupadora principal del *software* de navegació de l'Apollo. Durant els seus inicis en el projecte l'Apollo, es va adonar que el *software* no es prenia tan seriosament com altres disciplines de l'enginyeria dins i fora de la informàtica. Hamilton i el seu equip van crear mètodes, estàndards, regles i eines per desenvolupar el *software* de navegació. Per donar legitimitat al procés de construcció de *software*, va tenir la idea d'anomenar la disciplina *enginyeria del software*.

El curs 2017-2018 ha estat un any molt especial per a la FIB, ja que va celebrar el 40è aniversari. Des del 1977, la nostra escola ofereix ensenyament universitari de qualitat i excel·lència en l'àmbit de l'enginyeria informàtica, que respon a les necessitats de l'entorn social i econòmic. L'enginyeria del *software* i la programació amb criteris d'alta qualitat són elements bàsics de l'ensenyament a la FIB i constitueixen el domini de treball de diversos grups de recerca de la UPC. Per això, hem proposat aquesta candidatura i avui estem orgullosos de donar la benvinguda a la Sra.

Margaret H. Hamilton, pionera en la història de la informàtica, al Claustre de la UPC.

D'altra banda, la comunitat de l'enginyeria del *software* celebra el 50è aniversari de la disciplina enguany. Diverses institucions i conferències commemoren aquest fet arreu del món. Aquesta cerimònia de doctorat *honoris causa* és el nostre reconeixement de les contribucions de la Sra. Hamilton a aquesta disciplina, la disciplina a la qual ella va donar el nom.

A més, l'any vinent el món commemorarà el 50è aniversari de la primera arribada de l'home a la Lluna. Les contribucions de la Sra. Hamilton i el seu equip van ser crucials perquè l'Apollo 11 aterrés amb èxit a la Lluna el 20 de juliol de 1969. Per aquest assoliment històric, les escoles d'aeronàutica de la UPC, així com la FIB, estan orgullosos de donar-li la benvinguda com a nou membre del Claustre Universitari.

Certament, els èxits de la Sra. Margaret H. Hamilton han estat clau en els dos àmbits: l'enginyeria informàtica i l'enginyeria aeronàutica.

Margaret Heafield Hamilton, nascuda el 17 d'agost de 1936, a Paoli, Indiana, va estudiar matemàtiques a la Universitat de Michigan el 1954 i posteriorment es va llicenciar en matemàtiques, amb un mínor en filosofia, a l'Earlham College el 1958. Es va traslladar a Boston, Massachusetts, on el seu marit va anar a la Universitat Brandeis per fer un màster en química, seguit d'una llicenciatura en dret a l'Escola de Dret de Harvard. Durant aquest temps, Hamilton va treballar per donar suport a la seva família.

Hamilton volia fer estudis de postgrau en matemàtica abstracta a la Universitat Brandeis tan aviat com ella i el seu marit acabessin de fer torns per finalitzar els seus estudis en les escoles respectives. Esmenta una professora de matemàtiques com a mentora i com una influència important sobre el seu desig de continuar estudiant matemàtica abstracta. Va tenir altres inspiracions fora del món tecnològic, incloent-hi el seu pare, que era filòsof i poeta, i el seu avi, director d'escola i ministre quàquer. Diu que tant el seu pare com el seu avi la van inspirar per estudiar filosofia.

Per preparar-se per als estudis de postgrau en matemàtica abstracta a la Universitat Brandeis, el 1959 va prendre un càrrec provisional al MIT per desenvolupar *software* per a la previsió del temps amb els ordinadors del MIT LGP-30, que va programar en hexadecimal, i PDP-1, per al professor Edward Norton Lorenz al departament de meteorologia.

Margaret H. Hamilton va desenvolupar el *software* per a un sistema de defensa aèria semiautomàtic (SAGE) als laboratoris Lincoln. Tan aviat com va sentir que el MIT buscava gent per desenvolupar el *software* per enviar l'home a la Lluna, es va unir al projecte de la NASA i el MIT per desenvolupar el *software* de navegació per a les missions tripulades. En aquest moment primerenc, la informàtica i l'enginyeria del *software* encara no eren disciplines; els programadors havien d'aprendre sobre la marxa, adquirir experiència amb la pràctica.

Del 1961 al 1963, va treballar en el projecte SAGE als laboratoris Lincoln, on va ser una dels programadors que va escriure *software* per al primer ordinador AN/FSQ-7 (l'XD-1), per buscar avions enemics; també va escriure *software* per als Laboratoris de Recerca de les Forces Aèries a Cambridge.

El projecte SAGE va ser una extensió del projecte Whirlwind, que va iniciar el MIT, per crear un sistema informàtic que pogués predir els sistemes meteorològics i rastrejar-ne els moviments amb simuladors. Aviat es va desenvolupar el SAGE per a ús militar en la defensa antiaèria de possibles atacs soviètics durant la Guerra Freda. Van ser els seus esforços en aquest projecte que la van fer candidata al lloc a la NASA com a desenvolupadora principal del *software* de navegació de l'Apollo.

A continuació, Hamilton es va incorporar al laboratori Charles Stark Draper al MIT, on va començar a treballar en el *software* de navegació per a les missions no tripulades de l'Apollo. Poc

després, van posar Hamilton a càrrec de tot l'equip que va desenvolupar el *software* de navegació per a les missions tripulades de l'Apollo i les missions Skylab posteriors. Això incloïa el *software* desenvolupat a l'ordinador de navegació de l'Apollo per al mòdul de comandament (MC), el mòdul lunar (ML) i les interfícies ("l'adhesiu") entre totes les fases de la missió. També incloïa el *software* de sistemes, que es compartia entre i residia dins del MC i el ML. El *software* de sistemes incloïa el *software* de detecció i recuperació d'errors, com ara els reinicis i les visualitzacions de prioritat, que Hamilton va dissenyar i desenvolupar. No es van produir errors de *software* durant els vols reals.

Va treballar per adquirir experiència pràctica durant un temps en què els cursos d'informàtica eren molt escassos i no existien cursos d'enginyeria del *software*.

La missió Apollo 11 va ser especial. Cap humà mai no havia aterrat a la Lluna abans. Tot anava perfectament fins que va passar una cosa totalment inesperada, precisament en el moment crític just abans de l'aterratge. Tres minuts abans que el mòdul d'aterratge arribés a la superfície de la Lluna, l'ordinador de navegació de l'Apollo es va sobrecarregar.

Les alarmes del programa van indicar "executive overflows", que significava que l'ordinador de navegació no podia completar totes les seves tasques en temps real i havia d'ajornar-ne algunes.

Tanmateix, gràcies als sistemes d'alarma i de tasques prioritàries desenvolupats per l'equip de Hamilton, la situació es va gestionar i les pantalles de prioritat van donar els astronautes l'elecció d'aterrar o no aterrar. I van aterrar. Els tripulants de l'Apollo 11 es van convertir en els primers éssers humans a trepitjar la Lluna i el *software* de l'equip de Hamilton es va convertir en el primer *software* a executar-se a la Lluna.

El Dr. Paul Curto, tecnòleg sènior que va nominar Margaret Hamilton per al Premi Space Act de la NASA, va dir que el treball de Hamilton era "la base per al disseny de *software* ultrafiable".

El grup d'enginyeria del *software* de Hamilton va treballar en diversos projectes després de l'Apollo.

El 1976, Margaret H. Hamilton va cofundar una empresa anomenada Higher Order Software. Ella en va ser la directora general del 1976 al 1984. L'objectiu era desenvolupar les idees sobre la prevenció d'errors i la tolerància a fallades que van sorgir de la seva experiència al MIT.

Van crear un producte anomenat USE.IT, basat en la metodologia del *software* d'ordre superior (*higher-order software*, HOS) desenvolupada al MIT. Es va utilitzar amb èxit en nombrosos projectes governamentals. Un dels projectes destacats va consistir a formalitzar i implementar el primer IDEF (Integration DEFinition) computable, el C-IDEF, per a les forces aèries, sobre una base formal de HOS.

Una anàlisi detallada de la teoria de HOS i del llenguatge AXES va ser utilitzada per Harel per desenvolupar un llenguatge derivat per a una forma més moderna de programació estructurada derivada del HOS, anomenat *llenguatge de programació And/Or*, des del punt de vista dels subobjectius and/or.

Harel va mostrar com el HOS i el seu llenguatge de programació and/or derivat es relacionen amb la lògica matemàtica, la teoria del joc i la intel·ligència artificial. Altres han utilitzat el HOS per formalitzar la semàntica dels quantificadors lingüístics i formalitzar el disseny de sistemes incrustats en temps real fiables.

Margaret H. Hamilton va abandonar la companyia el 1985 i va continuar treballant en la recerca del *software* lliure d'errors.

El març de 1986, es va convertir en la fundadora i directora general de Hamilton Technologies, Inc., a Cambridge, Massachusetts. La companyia es va desenvolupar al voltant del llenguatge de sistemes universal (USL) i el seu entorn automatitzat associat, el 001 Tool Suite, basat en el paradigma de desenvolupament abans dels fets (DBTF) de Hamilton per al disseny

de sistemes i el desenvolupament de *software*. Hamilton Technologies i els seus clients van desenvolupar moltes aplicacions amb l'USL.

Margaret H. Hamilton va encunyar el terme *enginyeria del software* durant el temps que va treballar en les missions Apollo. En cinquanta anys, aquesta disciplina ha guanyat el mateix respecte que qualsevol altra disciplina de l'enginyeria.

Des del principi, com a enginyera del *software* pionera, estava molt preocupada pels possibles errors i les situacions inesperades. Durant la seva llarga carrera professional i de recerca, ha desenvolupat criteris, metodologies, llenguatges i eines per als sistemes ultrafiables. Ha publicat més de 130 articles, actes de congressos i informes sobre els 60 projectes i 6 programes significatius en què ha estat involucrada.

Els enginyers de Google reconeixen Margaret H. Hamilton com la primera enginyera de fiabilitat dels llocs (*site reliability engineer*).

Esmentaré alguns dels honors i premis que ha rebut: el 22 de novembre de 2016, va ser guardonada amb la Medalla Presidencial de la Llibertat pel president dels Estats Units, Barack Obama, pel seu treball liderant el desenvolupament del *software* de navegació de les missions Apollo a la Lluna de la NASA. Aquest és el màxim honor civil als Estats Units.

El 1986, va rebre el Premi Ada Lovelace de l'Association for Women in Computing. El 2003, va rebre el Premi Exceptional Space Act de la NASA per les contribucions científiques i tècniques. El guardó estava dotat amb 37.200 dòlars, la quantitat econòmica més gran atorgada a qualsevol persona en la història de la NASA. El 2009, va rebre el premi per a destacats antics alumnes de l'Earlham College.

El 28 d'abril de 2017, va rebre el Premi Fellow del Museu de la Història de l'Ordinador, que destaca dones i homes excepcionals les idees dels quals han canviat el món.

L'any 2017, es va posar a la venda un conjunt de LEGO anomenat Dones de la NASA, que conté (entre altres coses) petites figures de Margaret Hamilton, Mae Jemison, Sally Ride i Nancy Grace Roman.

Les contribucions de Margaret H. Hamilton han tingut un paper important per aconseguir que els éssers humans arribin a la Lluna, donar legitimitat a l'enginyeria del *software*, donar importància a la fiabilitat del *software* i ajudar a obrir la porta del camp de la informàtica a més dones.

Gràcies, Margaret, per totes les teves contribucions!

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.

Order of the award ceremony

Welcome from the rector of the Universitat Politècnica de Catalunya - BarcelonaTech, Prof. Francesc Torres.

Reading of the Governing Council's decision by the general secretary, Ms Marta de Blas.

Oration for Ms Margaret H. Hamilton by the sponsor, Prof. Núria Castell Ariño.

Conferral of the honorary doctorate on Ms Margaret H. Hamilton by the Universitat Politècnica de Catalunya · BarcelonaTech.

Acceptance speech by Ms Margaret H. Hamilton.

Speech by the rector of the UPC, Prof. Francesc Torres.

The music will be performed by
the EEBE's Gaudeamus choir,
the PMT's choir,
the FME's Ol·lari choir,
the Arquitectura choir and the UPC Orchestra;

Canticorum iubilo, by Georg Friedrich Händel

Cançó de bres per a una princesa negra, by A. Rodríguez Sabanes, arranged by D. Antolí

A round of three country dances in one, arranged by Thomas Ravenscroft

Gaudeamus igitur, arranged by Joan Casulleres

Gaudeamus igitur

Gaudeamus igitur,
iuvenes dum sumus. (bis)
Post iucundam iuventutem,
post molestam senectutem,
nos habebit humus.

Vivat academia,
vivant professores.
Vivat membrum quodlibet,
vivant membra quaelibet,
semper sint in flore.

Gaudeamus igitur,
iuvenes dum sumus. (bis)
Post iucundam iuventutem,
post molestam senectutem,
nos habebit humus.

Oration for Ms Margaret H. Hamilton

by the sponsor Prof. Núria Castell Ariño
Universitat Politècnica de Catalunya · BarcelonaTech

Distinguished rector of the Universitat Politècnica de Catalunya · BarcelonaTech, distinguished members of the University Senate and Board of Trustees, authorities and representatives of institutions and companies, professors, students, administrative staff, family and friends, and dear Ms Margaret H. Hamilton.

It is a great pleasure to be here today at this solemn ceremony to welcome to the University Senate a new doctor *honoris causa* of the Universitat Politècnica de Catalunya (UPC), pursuant to the Governing Council decision of 23 May 2017, sponsored by the Facultat d'Informàtica de Barcelona (FIB), of which I am proud to have been the dean. Margaret H. Hamilton's nomination was supported by UPC departments and schools and several internationally well-known experts in software engineering and computing history.

Margaret H. Hamilton has a long list of accomplishments. She is a computer scientist, a systems engineer, and a business owner and entrepreneur. She is also known as the woman who got man to the Moon. During her long research and professional career, her research interests have included systems and software with a preventative paradigm, formal theory, ultra-reliable systems, error detection and recovery, modelling and programming languages, reusability, correctness by built-in language properties, man-machine interfaces, open architectures, seamless integration, operating systems, automated life-cycle environments, software development productivity, systems design and software development within an asynchronous, distributed system-of-systems environment, and software life-cycle management.

Hamilton began working on several software projects at the Massachusetts Institute of Technology (MIT) at Boston, learning by herself about systems and software. From there she got a position at NASA as the lead developer for the Apollo on-board flight software. During the early days of Apollo, she realized that software was not taken as seriously as other engineering disciplines within and outside computing. Hamilton and her team created methods, standards, rules and tools for developing the flight software. To give legitimacy to the process of building software she came up with the idea of naming the discipline "software engineering".

The 2017-2018 academic year has been a very special year for the FIB. It was the celebration of its 40th anniversary. Since 1977 our

school offers top-quality university education and excellence in the area of computing engineering that responds to the needs of the social and economic environment. Software engineering and programming with high-quality criteria are basic elements of education at the FIB, and constitute the working domain of several UPC research groups. Because of this, we sponsored this candidature and today we are proud to welcome Ms Margaret H. Hamilton, a pioneer in computer science history, to the UPC Senate.

Furthermore, the software engineering community celebrates the 50th anniversary of the discipline this year. Around the world several institutions and conferences commemorate this fact. The present award ceremony is our recognition of Ms Hamilton's contributions to this discipline, the discipline which she named.

In addition, next year the world will commemorate the 50th anniversary of the first landing by man on the Moon. The contributions of Ms Hamilton and her team were crucial to Apollo 11's successful landing on the Moon on 20 July 1969. For that historical accomplishment, the UPC's aeronautics schools, as well as the FIB, are proud to welcome her as a new University Senate member.

Certainly, the accomplishments of Ms Margaret H. Hamilton have been key in both areas: computing engineering and aeronautical engineering.

Margaret Heafield Hamilton, born on 17 August 1936, in Paoli, Indiana, studied mathematics at the University of Michigan in 1954 and subsequently earned a B.A. in mathematics with a minor in philosophy from Earlham College in 1958. She moved to Boston, Massachusetts, where her husband went to Brandeis for a master's degree in chemistry followed by a law degree at Harvard Law School. During this time, Hamilton worked to support her family.

Hamilton's plan was to pursue graduate study in abstract mathematics at Brandeis University as soon as she and her husband took turns finishing their studies at their respective schools. She cites a female math professor as a mentor and major influence on her desire to continue in the study of abstract mathematics. She had other inspirations outside the technological world, including her father, philosopher and poet, and her grandfather, a school headmaster and Quaker minister. She says both her father and grandfather inspired her to minor in philosophy.

To prepare for further studies in abstract math at Brandeis University, in 1959 she took an interim position at MIT to develop software for predicting weather on MIT's LGP-30 computer, which she did in hexadecimal, and MIT's PDP-1 computer, for Professor Edward Norton Lorenz in the meteorology department.

Margaret H. Hamilton then developed software for the Semi-Automatic Ground Environment (SAGE) air defense system at Lincoln Laboratories. As soon as she heard MIT was looking for people to build software for sending man to the Moon, she joined NASA/MIT's project to build Apollo's on-board flight software for the manned missions. At that early time, computer science and software engineering were not yet disciplines; instead, programmers had to learn on the job with hands-on experience.

From 1961 to 1963, she worked on the SAGE Project at Lincoln Lab, where she was one of the programmers who wrote software for the first AN/FSQ-7 computer (the XD-1), to search for unfriendly aircraft; she also wrote software for the Air Force Cambridge Research Laboratories.

The SAGE Project was an extension of Project Whirlwind, started by MIT, to create a computer system that could predict weather systems and track their movements through simulators. SAGE was soon developed for military use in anti-aircraft air defense from potential Soviet attacks during the Cold War. It

was her efforts on this project that made her a candidate for the position at NASA as the lead developer for the Apollo on-board flight software.

Hamilton then joined the Charles Stark Draper Laboratory at MIT, where she began by working on flight software for the Apollo unmanned space missions. Shortly thereafter, Hamilton was put in charge of the whole team that developed the Apollo on-board flight software for the manned missions and the subsequent Skylab missions. This included the software developed on the Apollo Guidance Computer for the Command Module (CM), the Lunar Module (LM), and the interfaces ("glue") between and among all the mission phases. It also included the systems software, which was shared by and resided within both the CM and the LM. The system software included the error detection and recovery software such as the restarts and the Priority Displays, which Hamilton designed and developed. No software errors surfaced during actual flights.

She worked to gain hands-on experience during a time when computer science courses were very rare and software engineering courses did not exist.

The Apollo 11 mission was special. No human had ever landed on the Moon before. Everything was going perfectly until something totally unexpected happened, precisely at the critical moment just before landing. Three minutes before the Lunar lander reached the Moon's surface, the Apollo Guidance Computer became overloaded.

Program alarms indicated "executive overflows", meaning the guidance computer could not complete all of its tasks in real time and had to postpone some of them.

However, thanks to the alarm and priority tasks systems developed by Hamilton's team, the situation was managed, and the Priority Displays gave the astronauts the choice to land or not to land. And they landed. The Apollo 11's crew became the first humans to walk on the Moon, and the Hamilton team's software became the first software to run on the Moon.

Dr. Paul Curto, senior technologist who nominated Margaret Hamilton for a NASA Space Act Award, called Hamilton's work "the foundation for ultra-reliable software design".

Hamilton's software engineering group worked on several projects after Apollo.

In 1976, Margaret H. Hamilton co-founded a company called Higher Order Software. She was the CEO from 1976 through 1984. Their aim was to further develop ideas about error prevention and fault tolerance emerging from her experience at MIT.

They created a product called USE.IT, based on the Higher Order Software (HOS) methodology developed at MIT. It was successfully used in numerous government projects. One notable project was to formalize and implement the first computable IDEF (Integration DEFinition), C-IDEF for the Air Force, based on HOS as its formal foundation.

A detailed analysis of the HOS theory and the AXES language was used by Harel to develop a derived language for a more modern form of structured programming derived from HOS called the And/Or programming language from the viewpoint of and/or subgoals.

Harel goes on to show how HOS and his derived And/Or programming language relate to mathematical logic, game theory and artificial intelligence. Others have used HOS to formalize the semantics of linguistic quantifiers, and to formalize the design of reliable real-time embedded systems.

Margaret H. Hamilton left the company, HOS, in 1985. At this time, she continued her pursuit of error-free software.

In March 1986, she became the founder and CEO of Hamilton Technologies, Inc. (HTI) in Cambridge, Massachusetts. The company was developed around the Universal Systems Language (USL) and its associated automated environment, the 001 Tool Suite, based on her paradigm of Development Before The Fact (DBTF) for systems design and software development. HTI and its customers developed many applications using USL.

Margaret H. Hamilton coined the term "software engineering"-during her days working on the Apollo missions. In 50 years, this discipline has gained the same respect as any other engineering discipline.

From the very beginning, as a pioneer software engineer, she was very concerned about possible errors and unexpected situations. During her long research and professional career, she has developed criteria, methodologies, languages, and tools for ultra-reliable systems. She has published over 130 papers, proceedings, and reports about the 60 projects and 6 major programs in which she has been involved.

Google engineers credit Margaret H. Hamilton as the first Site Reliability Engineer (SRE).

To mention some of the honors and awards already presented to her: on 22 November 2016, Margaret H. Hamilton was awarded the Presidential Medal of Freedom by U.S. President Barack Obama for her work leading the development of the on-board flight software for NASA's Apollo Moon missions. This is the highest civilian honor in the United States.

In 1986, she received the Ada Lovelace Award by the Association for Women in Computing. In 2003, she was given the NASA Exceptional Space Act Award for scientific and technical contributions. The award included $37,200, the largest amount awarded to any individual in NASA's history. In 2009, she received the Outstanding Alumni Award from Earlham College.

On 28 April 2017, she received the Computer History Museum Fellow Award that honors exceptional men and women whose ideas have changed the world.

In 2017, a "Women of NASA" LEGO set went on sale featuring (among other things) mini-figurines of Margaret Hamilton, Mae Jemison, Sally Ride, and Nancy Grace Roman.

Margaret H. Hamilton's contributions have played an important role in getting humans to the Moon, in giving legitimacy to software engineering, in giving importance to software reliability, and in helping to open the door for more women to enter the computing field.

Thanks, Margaret, for all your contributions!

Acceptance speech by Ms Margaret H. Hamilton

A Preventative Paradigm for Systems and Software

My background in software has been anything but traditional. It began before the discipline of designing and developing software was a field. There were no courses available to understand what it was we were doing or how we should be doing it. Any prior «education», if you will, was a combination of learning from seemingly unrelated life and work experiences before and during college, majoring in math and minoring in philosophy in college, and working in the trenches for years on several and varied software applications that required doing things that had never been done before while at the same time coming up with solutions and associated rules and techniques for doing such things in the future. On hindsight, I think it all began when I was very young. I remember, as a child, the many philosophical «what if?», «why?» and «why not?» talks I had with my father and grandfather. Both taught me to observe and question everything until answers made sense; and I look back at how important these talks were for what was to become important for my work in the future.

I was fortunate to have had unique challenges, responsibilities and experiences in the workplace, even before high school. All because I needed to raise money for college. One experience, in particular, stands out. Before I was 15, I was hired by the owner of an abandoned copper mine, in the Upper Peninsula of Michigan, that had been converted into a storage place for bananas. He wanted to turn the mine into a tourist attraction where people would be taken on tours through the mine and given lectures on how native copper was mined (this was the only place in the world where native copper existed). He wanted me to help jumpstart his new tourist business. He had not gone past fourth grade, and he knew he did not know enough basics (like math) to do everything himself. He asked me to learn as much as I could about copper, become the mine's first tour guide, and hire and train people to work for me as guides. Our tourist business went from giving the tour to a couple of people a day to thousands of people a day. Not only was I in charge of the guides, I was also responsible for almost everything else connected to the mine, including starting and managing a gift shop, being in charge of the financials and running the business at large. During this time I learned quickly that if you made mistakes you did not allow yourself (or others) to ever make them again! After such an experience, when it later became time to become familiar with and responsible for doing other completely new and unfamiliar kinds of things without any prior background or experience, like designing and developing software, it was a non-issue; or so I thought at the time!

I realize also how fortunate I was to have had so many amazing experiences during the earlier days of software, when we were deep in the trenches, to be in a field before it became a field, and to be around those from whom I learned so much. In 1959, Edward N. Lorenz, a professor at MIT, introduced me to compu-

ters; I developed weather prediction applications in hexadecimal and binary on the LGP-30. Known for his ground-breaking work in chaos theory, he was a most humble human being. His love for software experimentation was contagious, and I caught the «bug». Towards this end, understanding the relationships between the hardware and the software was critical, because it was important on this project to develop software that was performance-optimized. We viewed errors, then, simply as a nuisance, especially since debug sessions took forever.

Another early project was SAGE (Semi-Automatic Ground Environment), for which I developed applications in assembly language on the first AN/FSQ-7 (the XD-1), at MIT's Lincoln Labs, to look for enemy airplanes. The machine was huge, the largest computer system ever built. It was especially important not to make an error, because if you did, the computer would tell on you with siren-like and fog horn-like sounds that everyone within a large warehouse-like building could hear, and flashing lights that everyone could see. Operators and programmers would come running to find out whose program had crashed. One time, in the middle of the night, I received a frantic call at home from one of the computer operators to tell me that something terrible was wrong with my program —it no longer sounded like a seashore. It then became clear that we had found a new way to debug, using sound!

Just as it was with the LGP-30, debugging on the AN/FSQ-7 on the SAGE project was time consuming, and tools did not exist for finding an error and that which made it happen. Software errors were unwelcome mostly because they were considered to be a nuisance (or an embarrassment). During this project, as with the LGP-30 project, I became fascinated with errors —looking for ways to understand what made a particular error(s) or class of errors happen and working on ways to prevent it from happening in the future.

Back then, you were on your own, and knowledge (or lack thereof) was passed down from person to person. A manager hired you if you «knew» the commands in his computer's native language. Like «knowing» a set of English words would mean you could write a novel. I was mystified by this attitude. If a system crashed, software was always the one to blame. Terms were undefined, leading to errors, misunderstandings and drama. Terms like «software», «design», «error», and «computer systems» meant different things to different people. In fact, one of the managers I worked for at first thought the term «software» meant «soft clothing». Also, at the time, the world of software was tribal. Software «types» often could mix things up if and when other software areas were unfamiliar to them (e.g., mixing up the operating system functionality with the target system functionality). Although some things were quite different then, the life cycle process itself by default was not unlike today's traditional life cycle (e.g., the waterfall, spiral, and agile models), going from requirements to coding to endless testing and maintenance.

Apollo On-Board Flight Software

Sage definitely came with drama, especially having to do with errors; but this was only the beginning of what would come next: the Apollo on-board flight software project at MIT, under contract to NASA. The challenge was that the software was man-rated, meaning astronauts' lives were at stake. It had to WORK —the first time. Not only did the software itself have to be ultra-reliable, but it would need to be able to detect an error and recover from it in real time. Learning was by «doing» and «being». Hardware engineers came with formal rules for designing and building hardware; for us, the «software engineers», this was not the case. Problems had to be solved that had never been solved before. At times, we made it up. Most developers were fearless and young, yet dedication and commitment were a given. Managers (mostly from hardware backgrounds) for whom software was a mystery gave us total freedom and trust. There was no time to be a beginner. I began by building software for the

unmanned missions, concentrating on the areas of the systems software, areas used by and impacting all of the flight software. The systems software included the software for error detection and recovery. The software system for the unmanned missions was synchronous.

Manned missions were next. I was now in charge of the team that developed the on-board flight software for the manned missions and the on-board flight software itself; but I made sure to keep my technical hand more often than not in the systems software area. The software was more complex for the manned missions. Our environment was asynchronous (a multi-programming environment), where higher priority jobs interrupted lower priority jobs based on every job's priority relative to every other job's priority. It was up to us, the developers, to manually assign a unique priority to every process in the flight software to ensure that all events would take place in the correct order and at the right time. I was always searching for new ways to prepare for and recover from the unexpected: going from program-specific to system-wide protection. I began to think about all of the possible «what-ifs» and worked on ways to address them. For example, I asked, what if there was an emergency during flight and we wanted to warn the astronauts? I thought there had to be a way to solve this. There was!

Each mission was exciting, but Apollo 11 was special. We had never landed on the moon before. Everything was going perfectly until something totally unexpected happened. Just as the astronauts were about to land on the moon, the flight computer became overtaxed! The software's Priority Displays (AKA Display Interface Routines) of 1201 and 1202 alarms interrupted the astronauts' normal mission displays to warn them there was an emergency, allowing NASA's Mission Control to understand what was happening and alerting the astronauts to place the rendezvous radar switch back in the right position. It quickly became clear that the software was not only informing everyone there was a hardware-related problem, but the software was also compensating for it. The Priority Displays gave the astronauts a go/no-go decision (to land or not to land). With only minutes to spare, the decision was made to go for the landing. The rest is history. The Apollo 11's crew became the first humans to walk on the moon, and our software became the first software to run on the moon.**[1, 2, 3]**

The software's systems-software error detection and recovery mechanisms had taken control. System-wide «kill and recompute» from a «safe place» snapshot-rollback restarts were triggered by the overloading, keeping only the highest priority jobs. During this moment, I remembered the most exciting and memorable discovery related to it. That was when the realization came to me that the steps taken earlier within the multi-programming environment could become the basis for solutions within a multi-processing environment. That is, even though there was only one process executing at one time within a multi-programming environment, other processes were waiting in parallel to that process. With this as a backdrop, the Priority Displays were able to be created, changing the interface between the flight software and the astronauts from synchronous to asynchronous (the software and astronauts becoming parallel processes within a system of systems). This would not have been possible without an integrated system of systems (and teams) approach and contributions made by other groups to support this becoming a reality. The hardware team at MIT changed their hardware and the mission planning team in Houston changed their astronaut procedures, both working closely with us to accommodate the Priority Displays for both the Command Module (CM) and the Lunar Module (LM), for any kind of emergency and throughout any mission. Mission Control was well prepared to know what to do if the Priority Displays interrupted the astronauts.

I thought of the years of preparing for this day and how fortunate I was to work with and share this experience with the many talented and dedicated people who made this possible. Coming up with discoveries, new ideas and solutions was an adventure.

From my own perspective, the software experience itself (designing it, developing it, evolving it, watching it perform and learning from it for future systems) was at least as exciting as the events surrounding the mission.

As developers, we had been given the opportunity of a lifetime: to make every kind of error humanly possible, each holding answers to questions we had not thought of asking. On hindsight, a blessing in disguise. The task at hand was to develop the CM and the LM software. This included the systems-software that resided within both the CM and the LM and was shared between them, and the structure of the software («glue») that defined the relationships between, among and within mission phases. Updates were continuously submitted from hundreds of people (including «guests») over time and the many releases for each and every mission (where the software for one mission was being worked on concurrently with the software for other missions). We had to make sure that everything would play together; that the software parts would successfully interface to and work together with each other as well as with other systems (including the hardware, peopleware and missionware).

We were handicapped by the hardware's time and space constraints, giving software «experts» the license to be creative. «Creative» code (i.e., tricky programming) was admired more than the number of lines of code a person wrote. Requirements were «thrown over the wall» by «non-software experts» who assumed that all the software programs would somehow «magically» interface together. Fortunately, this was not the case. For, if it had been, we would never have learned what was to come next. Further, because of the computer's constraints, data storage locations were shared between, among and within mission phases; and because of the multi programming environment, responsibilities and data storage locations were also shared among many programs continuously interrupting lower priority programs based on time and priority during each and every mission phase. This included both synchronous and asynchronous man-in-the-loop multi-processing within a system of systems. Although there were more than enough opportunities to make errors, there were now the opportunities to come up with solutions to prevent them. «Software engineering» rules evolved with each relevant discovery, while top management rules from NASA went from complete freedom to overkill. *Even though there was no lack of opportunity to make an error and just about any kind of error possible, no on-board flight software errors were ever known to occur during flight.*

Having been through these experiences, one could not help but do something about learning from them. We asked, «What can we do better for future systems? What should we keep doing because we are doing it right?» With initial funding from NASA and the DoD, we performed an empirical study of the Apollo effort. Our analysis took on multiple dimensions, not just for space missions but for systems in general. Lessons learned from this effort (and their impact) continue today: always ask "what if?" and always expect the unexpected. We learned that systems are asynchronous, distributed and event-driven in nature, and that this should be reflected in the language to define them and the tools to build them —characterizing their natural behavior in terms of real-time execution semantics. Having done so, there is no longer a need to explicitly define schedules of when events occur. By describing interactions between objects, the schedule of events is inherently defined. We also learned that the life cycle of a target system is a system with its own life cycle and that every system is inherently a system of systems.

Most interesting of all were the lessons we learned from the errors found during pre-flight testing. They were full of surprises. They told us what to do and where to go. After categorizing the errors, we concentrated our efforts on analyzing three categories of errors: interface errors (data, timing and priority conflicts), which were 75% of all the errors found, errors found by manual means with the Augekugel («eyeballing») or «scanning-listings» methods, and errors previously existing on earlier missions, which were usually the most subtle and hardest errors to

find.[4] Our analysis resulted in a theory based on lessons learned from Apollo and later projects. From its axioms, we derived a set of allowable patterns for defining a system. This led to a universal systems language (USL) together with its automation and preventative development paradigm, development before the fact (DBTF).[2,4,5,6,7,8,9]

It became clear one day that systems defined with the USL behaved differently than those defined with traditional languages. Of greatest interest was the realization that the root problem with traditional systems engineering and software development languages and their environments is that they support users in «fixing wrong things up» («after the fact») rather than in doing things in the right way in the first place («before the fact»). In contrast, with a preventative paradigm, instead of looking for more ways to test for errors, and continuing to test for errors late into the life cycle, the majority of errors, including all interface errors, are not allowed into a system in the first place, just by the way it is defined. Testing for non-existent errors becomes an obsolete endeavor.

We continue to discover new properties in systems defined with the USL. Each property «*comes along for the ride*» throughout a system's own development; the derivatives of the system's definition (its software being one of them) inherit the properties of the definition from which they are derived; integration from systems to software is seamless; reuse is inherent or derivable; a system defined with this language has properties in its definition that inherently support its own development «before the fact»; and[2] each of its systems has «built-in» reliability and «built-in» productivity throughout its life cycle. In contrast to the traditional paradigm with its «test to death» philosophy, with the preventative paradigm the more reliable the system, the higher the productivity in building it, minimizing the need for most testing. Much of the design and all of the code is automatically generated by the USL's automation, inheriting all of the properties of the definition from which it came. The developer doesn't ever need to manually code or manually change the code; only the changed part of the system is regenerated and integrated with the rest of the application, automatically.

We learned that, because the USL is syntax-, implementation-, and architecture-independent, its systems can be developed for diverse architectures, and that the syntax of other languages can be mapped to the underlying semantics of the USL.[7] Unlike a formal language that is mathematically based but limited in scope from a practical standpoint (e.g., kind or size of system), the USL extends traditional mathematics with a unique concept of control, enabling it to support the definition of any kind or size of system. With this paradigm, the language also provides input to and serves as the basis for its automation to inherit and pass on the definition's properties of control for the system's own development process. The USL's automation, a large system (millions of lines of code) in its own right, is completely defined with and generates itself.

It is not magic. These things are possible because of the USL's mathematical foundation. Not only does it take its roots from Apollo and later systems, it also takes roots from other formal methods, formal linguistics, and object technologies. Evolved over several decades, it has always stood its own in several different kinds of environments including academic[6,10,11], government agencies[12], commercial[8] and other environments.[13] Used in research and «trail blazer» organizations, it has been positioned for more widespread use. A radical departure from the traditional, it redefines what is possible. New to the world at large, it would be natural to make assumptions about what is possible and impossible based on its superficial resemblance to other languages —like traditional object-oriented languages. We have learned that it helps to suspend any and all preconceived notions when one is first introduced to this language, because it is a world unto itself —a different way to think about systems.

When today's developers of middle to large systems are asked to list their most pressing issues, this is what they say: integration

is too late, if at all; traceability, flexibility and evolvability are lacking; reuse is ad hoc and error-prone; software is unreliable even with extensive testing; software costs too much and it takes too long to make. Clearly, the last issues on the list, which have to do with reliability and productivity, become solved if most of, if not all of, the other issues on the list are solved. Most people say it is impossible to do much about addressing these issues —at least in the foreseeable future— and so software by its very nature is destined to have these kinds of problems. I ask, «Why is it that today's developers list the same issues as they listed 50 years ago when the field was brand new, and why is it that they give the same reason they gave 50 years ago?»

It is true that many of the pressing software issues that existed in the earlier days still exist today. What we have learned, however, from our own work, is that the reason they gave both then and now is not the one I would give. That is, I believe that the reason for the issues related to developing software is not the nature of software, per se, but it is instead largely because of the traditional paradigm used to build it —a paradigm that has been around since the beginning and continues in force to this day. Many of the well-known problems that exist with the traditional paradigm need no longer exist with a preventative paradigm. Moreover, what works best for developing ultra-reliable systems just happens to work best for systems in general, no matter the application. It has been shown that many aspects of the pressing issues (both 50 years ago and today) can be addressed; if not, ultimately eliminated altogether, with the use of the preventative paradigm.**[2,4,7,8,9,10,11,12,13]** The preventative paradigm with its language and its automation did not disappoint when put to test. In fact, the larger and the more complex the system, the better the results. What is able to be done with what has been learned, however, depends on how ready and how open developers are to a change in how we build software. Given the kinds of systems we need to build for today and tomorrow, I believe that change has to, and will begin to, happen, sooner rather than later.

For whatever success I may have experienced, much of it was because I was in the right place at the right time, with the right opportunities and the right people. In some ways, I had the benefit of beginning with no preconceived notions, since it was necessary to go into areas that had never been gone into before. Many of the things we were doing had not been done before and for that I feel very lucky. Much of the credit goes not only to those I have learned so much from, but also to the errors I have had the opportunity of having had some responsibility in their making, without which we would not have been able to learn the things we did. Some with great drama and fanfare, and often with a large enough audience to not want such a thing to ever happen again!

The errors showed us how to exist without them. They led to a language with a preventative paradigm where a system's definition *inherently replaces* much of what used to be aspects of the system's own life cycle, and which now becomes no longer needed; and it *serves as the input* for the language's automation of what used to be manual processes in the system's own life cycle, and therefore results in many parts of the system's own life cycle becoming no longer needed. In essence, trail blazing and taking risks in unknown territory led us among other things to the errors; the errors led us to a paradigm that leads «before the fact» to the future. Educating people how to think, do and be in

terms of the paradigm becomes the next real challenge.

References

[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.

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH